TURING 图灵程序设计丛书

[PACKT] PUBLISHING



[西] Viktor Farcic　Alex Garcia 著　袁国忠 译

Java 测试驱动开发

Test-Driven Java Development

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

作者简介

Viktor Farcic

资深软件架构师，Docker船长，Java开发专家，热衷于测试驱动开发、行为驱动开发、持续集成、持续交付和持续部署。

他把多年经验都分享在了博客上，深受读者欢迎：http://TechnologyConversations.com。

Alex Garcia

资深Java程序员，敏捷实践拥趸，热衷于学习新语言、新范式、新框架。

图灵程序设计丛书

Java 测试驱动开发

Test-Driven Java Development

[西] Viktor Farcic　Alex Garcia 著
袁国忠 译

人 民 邮 电 出 版 社
北　京

图书在版编目（C I P）数据

Java测试驱动开发 / (西) 维克多·法西克
(Viktor Farcic) ，(西) 阿列克斯·加西亚
(Alex Garcia) 著 ; 袁国忠译. -- 北京 : 人民邮电出
版社, 2017.9
（图灵程序设计丛书）
ISBN 978-7-115-46501-6

Ⅰ. ①J… Ⅱ. ①维… ②阿… ③袁… Ⅲ. ①JAVA语
言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第182978号

内 容 提 要

本书介绍如何将各种 TDD 最佳实践应用于 Java 开发，主要内容包括：用 Java 语言进行 TDD 会用到的各种工具和框架，所需环境搭建；通过实际应用程序，展示 TDD 优点及开发中应注意的主要问题；TDD 是如何通过模拟内部和外部依赖来提升速度的；如何重构既有应用程序；详细介绍所有 TDD 最佳实践。

本书适合所有 Java 开发人员，也适合用其他语言编程的程序员了解 TDD。

◆ 著　　　　[西] Viktor Farcic Alex Garcia
译　　　　袁国忠
责任编辑　陈　曦
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：13
字数：307千字　　　　2017年 9 月第 1 版
印数：1 – 3 500册　　　　2017年 9 月北京第 1 次印刷

著作权合同登记号　图字：01-2017-4857号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

前　　言

测试驱动开发面世已有一段时间，但依然未被很多人采用，因为它难以掌握。虽然理论很容易，但要熟练使用，必须经过大量实践。

多年来，本书作者一直在使用TDD，并试图将其经验传授给你。身为开发人员，他们深信学习编码实践的最佳方式是编写代码和不断练习，本书秉承的正是这种理念——通过练习诠释所有TDD概念。本书犹如一次旅行，期间，你有机会将各种TDD最佳实践应用于Java开发。这次旅行结束时，你将成为TDD黑带，你的软件开发工具中也会多一个法宝。

本书内容

第1章阐述我们的目标——成为拥有TDD黑带的Java开发人员。要想知道我们将去往何方，必须对一些描述旅程的问题进行讨论，并找到答案。

第2章介绍并安装本书将用到的所有工具和框架，再搭建所需的环境。对于每个工具和框架，都将通过代码说明其优缺点。

第3章演示如何使用TDD的支柱——“红灯–绿灯–重构”过程开发一个“井字游戏”。我们将编写测试并确定其失败；然后编写实现测试的代码，运行所有测试并确定其通过；最后，重构并完善代码。

第4章开发“遥控军舰”应用程序，以充分展示TDD在单元测试中的威力。你将学习单元测试到底是什么、它与功能测试和集成测试有何不同以及它在测试驱动开发中扮演的角色。

第5章以传统方法开发Connect4游戏。这个开发过程中，没有编写任何测试，而等到开发结束后才编写。通过这样做，你将认识到开发应用程序时，如果不采用使其易于测试的开发方法，将面临什么样的难题。

第6章阐述速度对TDD来说至关重要。为快速演示一些理念和概念，我们将扩展前面开发的“井字游戏”，并使用MongoDB存储数据。所有测试实际上都没有使用MongoDB，因为我们将模拟所有与MongoDB的通信。

第7章讨论如何使用BDD方法开发一个书店应用程序。我们将以BDD方式制定验收标准，分别实现各项功能。通过运行BDD场景确认每项功能都能正常工作，并在必要时重构代码使其达到预期的质量水平。

第8章介绍如何重构既有的应用程序。我们将首先为既有代码创建测试，然后不断重构，直到测试和代码都满足预期。

第9章演示如何开发一个斐波那契数列计算器，以及如何使用功能开关隐藏还未完成或出于商业考虑不应向用户发布的功能。

第10章详细介绍所有TDD最佳实践，并温习通过阅读本书获得的知识和经验。

需要什么

为完成本书的练习，读者必须有一台64位计算机。对于各种需要用到的软件，本书提供了详尽的安装说明。

为谁而写

如果你是经验丰富的开发人员，想学习更有效的系统和应用程序开发方法，那么本书就是为你而写的。

排版约定

为将不同类型的信息区分开来，本书使用了很多文本样式。下面列出其中一些样式及含义。

正文中的代码、数据库表名、文件夹名称、文件名、文件扩展名、路径名、URL、用户输入和Twitter账号，使用如下样式：

“通过使用指令`include`，可包含其他上下文。”

代码块使用如下样式：

```
public class Friendships {
    private final Map<String, List<String>> friendships = new
      HashMap<>();

    public void makeFriends(String person1, String person2) {
        addFriend(person1, person2);
        addFriend(person2, person1);
    }
```

命令行输入或输出使用如下样式：

```
$> vagrant plugin install vagrant-cachier
$> git clone thttps://bitbucket.org/vfarcic/tdd-java-ch02-example-
  vagrant.git
```

新术语和**重要词语**使用粗体。例如，对于出现在屏幕上的菜单或对话框中的词语，表示如下：

“输入查询后，将看到按钮**Go**，请单击。”

[警告或重要的注意事项。]

[提示和技巧。]

读者反馈

欢迎提供反馈，请将你对本书的看法告诉我们：哪些方面是你喜欢的，哪些方面你不喜欢。读者反馈对我们来说很重要，因为这可以帮助我们推出更符合读者需求的著作。

要给我们提供反馈，只需向feedback@packtpub.com发送电子邮件，并在邮件主题中指出书名。

如果你有擅长的主题，并有志于写书或撰稿，请参阅www.packtpub.com/authors的撰稿指南。

客户支持

购买本社图书后，你将获得各种帮助，让手中图书最大限度地发挥功效。

勘误

虽然我们力图让图书内容准确无误，但错误仍不可避免。如果你在本社图书中发现错误（包括正文和代码），请告诉我们，我们将感激不尽。你这样做不仅可以让其他读者免遭同样的挫折，还可帮助我们改进该书的后续版本。无论你发现什么错误，都请告诉我们。为此，可以访问http://www.packtpub.com/submit-errata，输入书名，单击链接**Errata Submission Form**，再输入错误详情。提交的勘误得到确认后，将被上传到我们的网站或添加到既有的勘误列表。

要查看已提交的勘误，请访问https://www.packtpub.com/books/content/support，并在搜索框中输入书名，**Errata**栏将列出你搜索的信息。

打击盗版

网上散布的盗版材料是各类媒体屡禁不绝的问题。在保护版权和许可方面，本社的态度非常严肃。如果你在网上看到本社作品的非法复制品，请马上把网址或网站名告诉我们，以便我们能够采取补救措施。

请通过copyright@packtpub.com与我们取得联系，并提供可疑的盗版材料链接。

感谢你为保护我们的作者提供的帮助，也十分感激对于我们提供有价值内容的能力给予的保护。

问题

只要有与本书相关的问题，都可通过questions@packtpub.com与我们联系，我们将尽力解决。

电子书

扫描如下二维码，即可购买本书电子版。

目　　录

第 1 章　为何要关心测试驱动开发 ………… 1

1.1　为何要使用 TDD …………………… 1

1.1.1　理解 TDD …………………… 3

1.1.2　红灯–绿灯–重构 …………………… 3

1.1.3　速度是关键 …………………… 4

1.1.4　TDD 并非测试方法 …………………… 4

1.2　测试 …………………… 5

1.2.1　黑盒测试 …………………… 5

1.2.2　白盒测试 …………………… 5

1.2.3　质量检查和质量保证的差别 ……… 6

1.2.4　更好的测试 …………………… 6

1.3　模拟 …………………… 7

1.4　可执行的文档 …………………… 7

1.5　无需调试 …………………… 9

1.6　小结 …………………… 9

第 2 章　工具、框架和环境 …………………… 10

2.1　Git …………………… 10

2.2　虚拟机 …………………… 11

2.2.1　Vagrant …………………… 11

2.2.2　Docker …………………… 13

2.3　构建工具 …………………… 14

2.4　集成开发环境 …………………… 15

2.5　单元测试框架 …………………… 16

2.5.1　JUnit …………………… 17

2.5.2　TestNG …………………… 19

2.6　Hamcrest 和 AssertJ …………………… 21

2.6.1　Hamcrest …………………… 21

2.6.2　AssertJ …………………… 22

2.7　代码覆盖率工具 …………………… 23

2.8　模拟框架 …………………… 24

2.8.1　Mockito …………………… 26

2.8.2　EasyMock …………………… 28

2.8.3　PowerMock …………………… 29

2.9　用户界面测试 …………………… 29

2.9.1　Web 测试框架 …………………… 30

2.9.2　Selenium …………………… 30

2.9.3　Selenide …………………… 31

2.10　行为驱动开发 …………………… 33

2.10.1　JBehave …………………… 33

2.10.2　Cucumber …………………… 35

2.11　小结 …………………… 37

第 3 章　红灯–绿灯–重构——从失败到成功再到完美 …………………… 38

3.1　使用 Gradle 和 JUnit 搭建环境 …………… 39

3.2　“红灯–绿灯–重构”过程 …………………… 41

3.2.1　编写一个测试 …………………… 41

3.2.2　运行所有测试并确认最后一个未通过 …………………… 41

3.2.3　编写实现代码 …………………… 42

3.2.4　运行所有测试 …………………… 42

3.2.5　重构 …………………… 42

3.2.6　重复 …………………… 43

3.3　“井字游戏”的需求 …………………… 43

3.4　开发“井字游戏” …………………… 43

3.4.1　需求 1 …………………… 44

3.4.2　需求 2 …………………… 49

3.4.3　需求 3 …………………… 52

3.4.4　需求 4 …………………… 57

3.5　代码覆盖率 …………………… 58

3.6　更多练习 …………………… 59

3.7 小结 ······ 60

第 4 章 单元测试——专注于当下而非过往 ······ 61

4.1 单元测试 ······ 61
4.1.1 何为单元测试 ······ 62
4.1.2 为何要进行单元测试 ······ 62
4.1.3 代码重构 ······ 62
4.1.4 为何不只使用单元测试 ······ 63
4.2 TDD 中的单元测试 ······ 64
4.3 TestNG ······ 64
4.3.1 注解@Test ······ 64
4.3.2 注解@BeforeSuite、@BeforeTest、@BeforeGroups、@After Groups、@AfterTest 和@AfterSuite ······ 65
4.3.3 注解@BeforeClass 和@AfterClass ······ 65
4.3.4 注解@BeforeMethod 和@AfterMethod ······ 66
4.3.5 注解参数@Test(enable = false) ······ 66
4.3.6 注解参数@Test(expected-Exceptions = SomeClass.class) ······ 66
4.3.7 TestNG 和 JUnit 差别小结 ······ 66
4.4 “遥控军舰”的需求 ······ 66
4.5 开发“遥控军舰” ······ 67
4.5.1 创建项目 ······ 67
4.5.2 辅助类 ······ 69
4.5.3 需求 1 ······ 69
4.5.4 需求 2 ······ 72
4.5.5 需求 3 ······ 74
4.5.6 需求 4 ······ 75
4.5.7 需求 5 ······ 77
4.5.8 需求 6 ······ 80
4.6 小结 ······ 81

第 5 章 设计——难以测试说明设计不佳 ······ 82

5.1 为何要关心设计 ······ 82
5.2 Connect4 ······ 84
5.3 完成 Connect4 实现后再测试 ······ 85
5.3.1 需求 1 ······ 85
5.3.2 需求 2 ······ 86
5.3.3 需求 3 ······ 87
5.3.4 需求 4 ······ 88
5.3.5 需求 5 ······ 89
5.3.6 需求 6 ······ 89
5.3.7 需求 7 ······ 90
5.3.8 需求 8 ······ 91
5.4 使用 TDD 实现 Connect4 ······ 92
5.4.1 Hamcrest ······ 92
5.4.2 需求 1 ······ 93
5.4.3 需求 2 ······ 93
5.4.4 需求 3 ······ 96
5.4.5 需求 4 ······ 97
5.4.6 需求 5 ······ 99
5.4.7 需求 6 ······ 99
5.4.8 需求 7 ······ 100
5.4.9 需求 8 ······ 101
5.5 小结 ······ 103

第 6 章 模拟——消除外部依赖 ······ 104

6.1 模拟 ······ 104
6.1.1 为何使用模拟对象 ······ 105
6.1.2 术语 ······ 106
6.1.3 模拟对象 ······ 106
6.2 Mockito ······ 107
6.3 “井字游戏”第二版的需求 ······ 107
6.4 开发“井字游戏”第二版 ······ 107
6.4.1 需求 1 ······ 108
6.4.2 需求 2 ······ 118
6.5 集成测试 ······ 124
6.5.1 分离测试 ······ 124
6.5.2 集成测试 ······ 125
6.6 小结 ······ 127

第 7 章 BDD——与整个团队协作 ······ 128

7.1 不同规范 ······ 128
7.1.1 文档 ······ 129

7.1.2 供程序员使用的文档 ……129
7.1.3 供非程序员使用的文档 ……130
7.2 行为驱动开发 ……130
7.2.1 叙述 ……131
7.2.2 场景 ……132
7.3 书店应用程序的 BDD 故事 ……133
7.4 JBehave ……136
7.4.1 JBehave 运行器 ……136
7.4.2 待定步骤 ……137
7.4.3 Selenium 和 Selenide ……138
7.4.4 JBehave 步骤 ……139
7.4.5 最后的验证 ……144
7.5 小结 ……146

第 8 章 重构遗留代码——使其重焕青春 ……147

8.1 遗留代码 ……147
8.2 编码套路 ……156
8.2.1 遗留代码处理套路 ……157
8.2.2 描述 ……157
8.2.3 技术说明 ……157
8.2.4 添加新功能 ……157
8.2.5 黑盒测试还是尖峰冲击测试 ……157
8.2.6 初步调查 ……158
8.2.7 应用遗留代码修改算法 ……161
8.2.8 提取并重写调用 ……166
8.2.9 消除状态的“基本类型偏执”坏味 ……170
8.3 小结 ……173

第 9 章 功能开关——将未完成的功能部署到生成环境 ……175

9.1 持续集成、持续交付和持续部署 ……175
9.2 功能开关 ……177
9.3 功能开关示例 ……178
9.3.1 实现 fibonacci 服务 ……181
9.3.2 使用模版引擎 ……184
9.4 小结 ……187

第 10 章 综述 ……188

10.1 TDD 概要 ……188
10.2 最佳实践 ……189
10.2.1 命名约定 ……189
10.2.2 流程 ……191
10.2.3 开发实践 ……192
10.2.4 工具 ……195
10.3 这只是开始 ……196
10.4 这并非终点 ……196

第 1 章 为何要关心测试驱动开发

本书的作者是开发人员，针对的读者也是开发人员，因此大部分学习都将通过代码进行。每章都将介绍一个或多个TDD实践，读者将通过完成套路掌握它们。在空手道中，套路（kata）是一种练习，学习者不断重复同样的招式，每次重复都进步一点点。同样，读者阅读每章后，都将有细微但意义重大的进步。你将学习如何改善设计和代码、缩短上市时间、提供与时俱进的文档、通过质量测试提高代码覆盖率以及编写行之有效的清晰代码。

旅程都有起点，本书也不例外。我们的目标是让你成为拥有**测试驱动开发**（**TDD**）黑带的Java开发人员。

为确定我们将走向何方，必须就一些决定航程的问题进行讨论并找到答案。何为TDD？这是一种测试方法还是别的什么东西？使用TDD有何好处？

本章旨在提供针对TDD的整体概括，帮你了解TDD定义及其优势。

本章涵盖如下主题：

- 理解TDD；
- 何为TDD；
- 测试；
- 模拟；
- 可执行的文档；
- 无需调试。

1.1 为何要使用 TDD

你所处的环境使用的可能是敏捷开发方法，也可能是瀑布开发方法；你们公司可能有明确的规程，这些规程经过了多年艰苦奋斗的洗礼；也可能，你们只是一家刚刚起步的创业公司。无论如何，你都很可能面临过下述一个乃至更多痛点、问题或导致交付失败的原因：

- 部分团队成员无缘参与需求、规范或用户故事的制定；
- 大部分乃至全部测试都是手动的，抑或根本就没有测试；
- 虽然使用了自动化测试，但并未检测出真正的问题；
- 编写并执行自动化测试的时间太晚，无法给项目带来真正的价值；
- 总是有更紧急的问题需要处理，没法腾出专门用于测试的时间；
- 整个团队分为测试、开发和功能分析小组，而这些小组常常不能同步；
- 无法重构代码，因为担心这样做会破坏既有的功能；
- 维护成本高；
- 上市时间过长；
- 客户觉得交付的产品不符合要求；
- 文档从来都不是最新的；
- 害怕部署到生产环境，因为结果无法预料；
- 常常无法部署到生产环境，因为运行回归测试的时间太长；
- 团队为搞清楚某些方法或类的作用花费的时间太多。

测试驱动开发并不能神奇地解决所有这些问题，而只为我们找到解决方案指明方向。世上没有灵丹妙药，但如果有什么开发实践能让众多层面的情况大不相同，那就是TDD。

测试驱动开发可缩短上市时间、简化重构工作、帮助创建更好的设计以及降低耦合程度。

除这些直接的好处外，TDD还是众多其他实践（如持续交付）的前提条件。使用TDD可改善设计和代码的质量、缩短上市时间、确保文档最新、获得极高的代码覆盖率等。

要掌握TDD并不那么容易。即便学习了所有的理论，仔细研究了最佳实践和反模式，旅程也才刚刚开始。要掌握TDD需要很长的时间和大量的实践，这是漫长的过程，绝不是阅读完本书就能结束的。事实上，这个过程根本就没有结束的时候，因为总是有新的方式面世，让你能够更熟练、更快捷地使用TDD。然而，需要付出的代价虽然很高，但带来的好处更多。使用TDD的时间足够长的人都宣称没有其他开发软件的方法，我们就是这样的人，你肯定也会成为其中一员。

学习编码技巧的最佳方式是实践，我们对此深信不疑。要掌握本书介绍的内容，仅在上班的路上翻阅还不够；这不是一本适合躺在床上阅读的书，你必须撸起袖子动手编写代码。

本章将介绍基础知识，但从下一章开始，你将通过阅读、编写和运行代码进行学习。我很想说等阅读完本书后，你就是经验丰富的TDD程序员了，但情况不是这样的。读完本书，你将熟悉TDD，并拥有坚实的理论和实践基础，而剩下的事就全靠你自己了。要想获得更多TDD经验，你必须在日常工作中使用TDD。

1.1.1　理解 TDD

此时你可能正自言自语："我知道TDD会带来一些好处，但测试驱动开发到底是什么呢？"TDD是一种简单的流程，要求你先编写测试，再编写实现代码，这与"编写代码后再测试"的传统方法相反。

1.1.2　红灯–绿灯–重构

测试驱动开发是一个过程，依赖于不断重复极短的开发周期。它基于**极限编程**（**XP**）的测试优先理念，倡导采用可高度信赖的简单设计。驱动这个流程前行的开发周期称为"**红灯–绿灯–重构**"。

这种流程本身很简单，由几个反复进行的步骤组成：

(1) 编写一个测试；
(2) 运行所有测试；
(3) 编写实现代码；
(4) 运行所有测试；
(5) 重构；
(6) 运行所有测试。

鉴于测试是在实现前编写的，因此它应该不能通过。如果通过了，就说明测试是错误的：要么它描述的功能早已存在，要么编写不正确。编写测试期间处于绿灯状态昭示着存在错报的问题，对于这样的测试，应将其删除或进行重构。

> 编写测试时，应处于红灯状态。完成测试要求的实现后，所有测试都应通过，此时将处于绿灯状态。

如果最后一个测试未通过，就说明实现不正确，必须修正：要么这个测试不正确，要么实现代码不符合我们制定的规范。如果其他测试未通过，就说明我们破坏了某种功能，必须撤销所做的修改。

在这种情况下，一种自然而然的反应是：花足够的时间修复代码，让所有测试都通过。然而，这样的做法是错误的。如果不能在几分钟内完成修复，最佳的选择是撤销所做的修改。毕竟修改前一切都正常，带来破坏的实现显然是错误的。为何不到原来的地方，重新考虑实现测试的正确方式呢？这样我们只是在错误的实现上浪费了几分钟，而不会为修复一开始就不正确的东西浪费更多时间。原有的测试覆盖率（不包括最后一个测试的实现）应该很高。我们通过有意识地重构来修改既有代码，而不将其作为修复最近编写的代码的方式。

> 不要试图让最后一个测试的实现完美无缺，而应只编写足以让这个测试通过的代码。

你可以任何喜欢的方式编写代码，但要快。一旦进入绿灯状态，我们就知道存在一个由测试构成的安全网，可接着重构代码了：改进和优化代码，但不引入新功能。重构结束后，所有测试应当在任何情况下都能通过。

如果重构期间有测试未通过，就说明重构破坏了既有功能，应像以前一样撤销所做的修改。在重构阶段，我们不修改任何功能，也不引入新的测试，而只改进代码，并不断运行所有测试，确保没有破坏任何功能。与此同时，我们证明了代码是正确的，并降低了未来的维护成本。

重构结束后，再重复整个过程。这是一个无限循环，每次循环都是一个极短的周期。

1.1.3　速度是关键

想想打乒乓球的情形吧。这项运动的节奏非常快，职业选手玩起来可能让人目不暇接，TDD与这项运动很像。TDD老手通常不会让接球（编写测试或实现的）时间超过一分钟：编写简短的测试并运行所有测试（乒），编写实现并运行所有测试（乓），再编写一个测试（乒），编写该测试的实现（乓），重构并确认所有测试都通过（计分）；然后重复上述过程：乒、乓、乒、乓、计分。不要试图让代码完美无缺，而应力图让球不断运动，直到需要计分（重构）为止。

> 测试和实现的切换时间应以分钟甚至秒计。

1.1.4　TDD 并非测试方法

TDD中的T常常遭人误解。测试驱动开发是一种设计方法，要求在编写代码前考虑实现以及代码需要提供的功能，且每次只关注一项功能的需求和实现——这有助于理清思路以及更好地组织代码。这并不意味着使用TDD时编写的测试毫无用处，甚至恰好相反：它们很有用，让我们能够以极快的速度进行开发，同时不担心破坏既有功能。对重构来说这显得尤为重要：能够在不担心破坏既有功能的情况下重新组织代码对改善质量大有裨益。

> 测试驱动开发的主要目标是提供可测试的代码设计，测试只是一项很有用的副产品。

1.2 测试

虽然测试驱动开发主要是一种代码设计方法，但测试也是其中一个很重要的方面，因此我们必须对如下两种测试方法有清晰的认识：

- 黑盒测试；
- 白盒测试。

1.2.1 黑盒测试

黑盒测试（也叫功能测试）将受测软件视为一个黑盒，无需知道其内部构造。这种测试是通过软件界面进行的，旨在确认它们像预期的那样工作。只要界面的功能未变，测试就应通过——即便内部构造发生了变化。测试人员知道程序该做什么，但不知道它是如何做的。黑盒测试是传统组织最常使用的测试类型。这种组织通常将测试人员划归到一个独立的部门——在测试人员不熟悉编程、难以理解代码时尤其如此。这种测试方法提供了外部观察受测软件的结果。

下面是黑盒测试的一些优点：

- 可高效测试大块代码段；
- 无需访问和理解代码，也不要求测试人员知道如何编写代码；
- 将用户角度和开发人员角度分离。

下面是黑盒测试的一些缺点：

- 覆盖率有限，因为只执行部分测试场景；
- 测试效率低下，因为测试人员对软件内部构造一无所知；
- 测试缺乏针对性，因为测试人员对应用程序的了解有限。

用于驱动开发的测试通常是根据验收标准进行的，而验收标准决定了要开发哪些功能。

> 自动化黑盒测试依赖于某种形式的自动化，如**行为驱动开发**（**BDD**）。

1.2.2 白盒测试

白盒测试（也叫透明盒测试、玻璃盒测试和结构测试）查看受测软件的内部，并将由此获得的知识用于测试过程。例如，如果在特定条件下应引发异常，可能需要在测试中重现这种条件。白盒测试要求测试人员了解系统的内部结构，同时具备编程技能；它提供了从内部观察受测软件的结果。

下面是白盒测试的一些优点：

- 可高效找出错误和问题；
- 知道受测软件的内部构造有助于进行详细测试；
- 能够发现隐藏的错误；
- 可帮助程序员反省；
- 有助于优化代码；
- 由于知道软件的内部构造，因此可最大限度地提高测试覆盖率。

下面是白盒测试的一些缺点：

- 可能无法发现未实现或缺失的功能；
- 需要对受测软件的内部构造有大致认识；
- 需要访问代码；
- 测试通常与产品代码的实现细节紧密耦合，导致重构代码后原本应该通过的测试未能通过。

白盒测试几乎都是自动化测试，且在大多数情况下都是单元测试。

> 在实现前执行的白盒测试是以**TDD**方式编写的。

1.2.3　质量检查和质量保证的差别

还可根据要达成的目标对测试方法进行分类。要达成的目标通常有两种：**质量检查（QC）**和**质量保证（QA）**。质量检查的重点是发现缺陷，而质量保证力图将缺陷消灭在萌芽状态。QC是面向产品的，旨在确保结果符合预期；而QA更专注于过程以确保制造质量，即力图确保以正确的方式做正确的事情。

> 质量检查过去扮演的角色更重要，但随着TDD、**验收测试驱动开发**（**ATDD**）和**行为驱动开发**（**BDD**）的面世，重点正转向质量保证。

1.2.4　更好的测试

无论使用黑盒测试、白盒测试还是两者兼而有之，编写测试的顺序都非常重要。

需求（规范和用户故事）是在实现需求的代码之前编写的，因此是它们定义了代码，而不是相反。对测试来说亦如此。如果它们是在代码之后编写的，那么从某种意义上说，是代码（及其实现的功能）定义了测试。由既有应用程序定义的测试有失偏颇，倾向于确认代码的功能，而不

是检查客户的期望是否得到满足，或者说代码的行为是否符合预期。如果是手动测试，这种倾向可能不那么严重，因为手动测试通常由独立的QC（即使通常称为QA）部门来做。这种部门以独立于开发人员的方式定义测试，这将导致更严重的问题，因为必然会出现部门间沟通不畅和“警察综合征”的问题。所谓“警察综合征”是指，测试人员不力图去帮助开发团队编写有质量保证的应用程序，而只会在流程结束后找茬。问题发现得越早，为修复而付出的代价越低。

> 以TDD（包括其ATDD和BDD等变种）编写的测试旨在未雨绸缪，将问题消灭在萌芽状态，确保开发的应用程序从根本上有质量保证。

1.3　模拟

要让测试能够快速运行并不断提供反馈，必须以合适的方式组织代码，以便能够轻松使用模拟对象（mock）和存根（stub）替换方法、函数和类。这种替换实际代码的方式通常称为“测试替身”。外部依赖可能严重影响执行速度，例如，代码可能需要与数据库通信。通过模拟外部依赖，可大幅提高速度。整个单元测试集的执行时间应以分钟乃至秒计。要想轻松使用模拟对象和存根，必须分离关注点以优化代码结构。

除可提高速度外，消除外部依赖还有其他更重要的好处。代码的外部依赖可能包括数据库、Web服务器、外部API等，这些外部依赖不但不可靠，而且访问需要很长时间。在很多情况下，这些外部依赖还可能不是现成的，例如，你可能需要编写与数据库通信的代码，并让人创建数据库模式（schema）。如果不使用模拟对象，就只能等到模式就绪后再测试。

> 无论是否使用模拟对象，都应以合适的方式编写代码，以便能够轻松用一个依赖对象替换另一个依赖对象。

1.4　可执行的文档

TDD（以及更多结构良好的测试）另一个很有用的方面是文档。要搞清楚代码是干什么的，在大多数情况下通过查看测试比查看实现本身要容易得多。一些方法的作用是什么？查看与之相关联的测试。应用程序某部分UI的功能是什么？查看与之相关联的测试。以测试方式编写的文档是TDD的支柱之一，有必要更深入地了解。

传统软件文档存在的主要问题是，它们通常都不是最新的。一部分代码发生变化后，文档便不再反映实际情况。几乎任何类型的文档都如此，需求和测试用例受到的影响最大。

需要为代码编写文档通常意味着代码本身写得不好。另外，不管你如何努力，文档都必然会过期。

开发人员不应依赖于系统文档，因为它几乎在任何时候都不是最新的。另外，在详尽而及时地描述代码方面，没有任何文档比代码本身做得更好。

将代码用作文档并不意味着不能有其他类型的文档，关键是避免重复。如果说通过阅读代码可获悉系统细节，那么其他类型的文档可提供快速指南和概述。非代码文档应回答诸如“系统的总体目标是什么”“系统使用了哪些技术”等问题。大多数情况下，简单的**README**足以提供开发人员所需的快速入门指南；对新来者而言，项目描述、环境搭建、安装以及构建和打包说明等部分很有用。至于其他方面，代码就是“圣经”。

实现代码提供了所需的所有细节，而测试代码描述了产品代码背后的意图。

> 测试就是可执行的文档，而TDD是创建和维护这种文档的最常用方式。

采用了某种**持续集成**（**CI**）时，不正确的测试文档将失败并迅速得到修复。CI能够解决测试文档不正确的问题，但无法确保所有功能都有相关文档。有鉴于此（以及众多其他原因），应以TDD的方式创建测试文档。如果编写实现代码前，所有功能都以测试的方式做了定义，且所有测试都通过，测试便提供了完整而最新的信息，可供开发人员使用。

对于团队的其他成员，该如何办呢？测试人员、客户、产品经理等都不是程序员，可能无法从产品代码和测试代码获取所需的信息。

前面说过，最常见的两种测试是黑盒测试和白盒测试。这种划分很重要，因为这也将测试人员分成了两类：知道如何编写或至少阅读代码的（白盒测试），不知道如何编写和阅读代码的（黑盒测试）。有些测试人员两种测试都能做，但通常都不知道如何编写代码，因此对开发人员来说很有用的文档对他们来说毫无用处。如果需要将文档与代码分开，单元测试并不是很好的选择，这正是BDD横空出世的原因之一。

> BDD可在保留TDD和自动化的优点的同时，提供非程序员所需的文档。

客户必须能够定义系统的新功能，还必须能够获取有关系统各重要方面的最新信息。因此文档的技术性不能太强（将代码作为文档不可行），同时必须在任何情况下都是最新的。BDD叙述（narrative）和场景（scenario）是提供这种文档的最佳方式之一。BDD故事可作为**验收标准**（在代码之前编写的），可频繁执行（最好每次提交时都执行），还是使用自然语言编写的，因此不但在任何情况下都是最新的，而且可供那些不想研究代码的人使用。

文档是软件不可分割的一部分，与代码一样需要经常测试，这样才能确保它既准确又是最新的。

> 要提供既准确又是最新的信息，唯一划算的方式是使用可集成到持续集成系统的可执行文档。

作为一种方法论，TDD提供了实现这种目标的良好途径。在底层，单元测试是最佳的可执行文档；而在功能层面，BDD是提供可执行文档的不错方式，它使用自然语言，确保了这种文档易于理解。

1.5　无需调试

作者几乎从未调试过自己编写的应用程序！这好像不可思议，但情况确实如此。我们几乎从不调试应用程序，因为没有理由这样做。在编写代码前编写测试且代码覆盖率很高的情况下，我们完全可以相信应用程序将像预期的那样工作。这并不意味着使用TDD编写的应用程序没有bug——bug肯定是有的，所有应用程序都有bug；但出现bug时，可轻松地找出它们——只需查看未被测试覆盖的代码即可。

测试本身可能没有涵盖某些情形。在这种情况下，应对措施是编写额外的测试。

> 代码覆盖率很高的情况下，与逐行调试直到找到罪魁祸首相比，通过测试找出导致bug的原因要快得多。

1.6　小结

通过阅读本章，你对测试驱动开发实践有了大致的认识，还知道了什么是真正的TDD。你了解到TDD是一种代码设计方法，这是通过简短而可重复的“红灯–绿灯–重构”周期进行的。

整个TDD过程中，“测试未通过”都是一种意料之中的状态，你不但应该欣然接受，还需想办法进入这种状态。“红灯–绿灯–重构”周期很短，从一个阶段切换到另一个阶段的速度极快。

虽然主要目标是代码设计，但TDD过程中创建的测试是宝贵的财产，应充分利用，它们还会严重影响我们对传统测试实践的看法。对于这些实践，我们对其中最常用的几个（如黑盒测试和白盒测试）做了简单的介绍，试图从TDD的角度审视它们，并指出了它们带来的好处。

你发现模拟对象是非常重要的工具，对编写测试来说常常必不可少。最后，我们讨论了可以也应该将测试用作可执行的文档，还有TDD如何极大地减少了调试的必要性。

介绍必要的理论知识后，下面搭建开发环境、概述并比较各种测试框架和工具。

第 2 章

工具、框架和环境

“你看到的是什么，你就是什么样的人；我们塑造工具，然后又为工具所塑造。”

——马歇尔·麦克卢汉

每位战士都熟悉自己的武器，同样，程序员必须熟悉开发生态环境以及简化编程的工具。无论你是否在工作中或家里使用过这些工具，都有必要重新审视它们，并比较其功能和优缺点。下面概述这些工具，并通过创建小型项目熟悉其中的几个。

后文将详细介绍这些工具和框架，此处不再赘述。本章的目标是热身，简要介绍它们的功能和工作原理。

本章涵盖如下主题。

- Git；
- 虚拟机；
- 构建工具；
- 集成开发环境；
- 单元测试框架；
- 代码覆盖率工具；
- 模拟框架；
- 用户界面测试；
- 行为驱动开发工具、框架和环境。

2.1 Git

Git是最受欢迎的修订控制系统，有鉴于此，本书使用的所有代码都存储在Bitbucket（https://bitbucket.org/）。如果你还没有安装Git，请现在就下载安装。它提供了用于各种流行操作系统的版本，这些都可在http://git-scm.com找到。

很多图形用户界面支持Git，其中包括Tortoise（https://code.google.com/p/tortoisegit）、Source

Tree（https://www. sourcetreeapp.com）和Tower（http://www.git-tower.com/）。

2.2　虚拟机

虽然虚拟机不在本书讨论范围之内，但这是一个功能强大的工具，对良好的开发环境来说不可或缺。它们在隔离的系统中提供易于使用的动态资源，让你能够根据需要使用和删除。这让开发人员能够专注于手头的任务，而不将时间浪费在从头创建或安装所需的服务上。这正是虚拟机能够在本书找到用武之地的原因：使用它们可让你专注于代码。

为确保无论哪种操作系统都能搭建出相同的环境，我们将使用Vagrant创建虚拟机，并使用Docker部署所需的应用程序。介绍操作方法时，我们将以Ubuntu操作系统为例，因为它是一种流行而常见的类UNIX系统；其中涉及的大多数技术都独立于平台，但有些情况下，你不能按这里介绍的做，因为你使用的可能是其他操作系统。这种情况下，你需要找出Ubuntu与你使用的操作系统的不同之处，并采取相应的措施。

2.2.1　Vagrant

我们将使用Vagrant创建开发环境栈，它提供了预置的盒子（box），能够轻松初始化现成的虚拟机。所有盒子和配置都放在名为**Vagrantfile**的文件中。

下面的示例创建一个简单的Ubuntu盒子，我们在其中添加了使用Docker安装MongoDB的配置（稍后将介绍如何使用Docker）。假设你在计算机中安装了VirtualBox（https://www.virtualbox.org）和Vagrant（https://www.vagrantup.com），还能访问互联网。

这个示例中，我们使用Ubuntu盒子（`ubuntu/trusty64`）创建了一个64位的Ubuntu实例，并将这个虚拟机的内存设置为1 GB：

```
config.vm.box = "ubuntu/trusty64"

config.vm.provider "virtualbox" do |vb|
vb.memory = "1024"
end
```

接下来，我们在这个Vagrant虚拟机中暴露MongoDB默认使用的端口，并使用Docker运行MongoDB：

```
config.vm.network "forwarded_port", guest: 27017, host: 27017
config.vm.provision "docker" do |d|
  d.run "mongoDB", image: "mongo:2", args: "-p 27017:27017"
end
```

最后，为提高Vagrant的安装速度，我们将缓存一些资源。为此，需要安装插件**cachier**，有

关该插件的更详细信息，请参阅https://github.com/fgrehm/vagrant-cachier。

```
if Vagrant.has_plugin?("vagrant-cachier")
  config.cache.scope = :box
end
```

下面启动这个虚拟机。首次启动虚拟机时，通常需要几分钟，因为需要下载并安装基本盒子和所有的依赖项：

```
$> vagrant plugin install vagrant-cachier
$> git clone thttps://bitbucket.org/vfarcic/tdd-java-ch02-example-vagrant.git
$> cd tdd-java-ch02-example-vagrant
$> vagrant up
```

运行这个命令时，输出如下：

```
vfarcic@viktor:~/IdeaProjects/tdd-java-ch02-example-vagrant$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/trusty64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/trusty64' is up to date...
==> default: A newer version of the box 'ubuntu/trusty64' is available! You currently
==> default: have version '20150609.0.7'. The latest is version '20150609.0.10'. Run
==> default: `vagrant box update` to update.
==> default: Setting the name of the VM: tdd-java-ch02-example-vagrant_default_1435347519969_47040
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 27017 => 27017 (adapter 1)
    default: 22 => 2222 (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection timeout. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if its present...
    default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Mounting shared folders...
    default: /vagrant => /home/vfarcic/IdeaProjects/tdd-java-ch02-example-vagrant
    default: /tmp/vagrant-cache => /home/vfarcic/.vagrant.d/cache/ubuntu/trusty64
==> default: Configuring cache buckets...
==> default: Running provisioner: docker...
    default: Installing Docker (latest) onto machine...
    default: Configuring Docker to autostart containers...
==> default: Starting Docker containers...
==> default: -- Container: mongoDB
==> default: Configuring cache buckets...
vfarcic@viktor:~/IdeaProjects/tdd-java-ch02-example-vagrant$ █
```

请耐心等待执行完毕。这个命令执行完毕后，你就有一个新的虚拟机，它使用的是Ubuntu操作系统，安装了`Docker`，并运行着一个`MongoDB`实例。最重要的是，所有这些都是使用一个命令实现的。

要查看当前运行的VM的状态，可使用参数status：

```
$> vagrant status
Current machine states:
default                    running (virtualbox)
```

要访问虚拟机，可使用SSH，也可使用Vagrant命令，如下所示：

```
$> vagrant ssh
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-46-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

 System information disabled due to load higher than 1.0

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

vagrant@vagrant-ubuntu-trusty-64:~$
```

最后，要让虚拟机停止运行，可先退出，再执行命令vagrant halt：

```
$> exit
$> vagrant halt
==> default: Attempting graceful shutdown of VM…
$>
```

有关详尽的Vagrant盒子列表以及如何配置Vagrant的更详细信息，请参见https://www.vagrantup.com。

2.2.2 Docker

搭建好环境后，安装所需的服务和软件。为此可使用Docker，它提供了一种在隔离的容器中安装并运行众多应用程序的便捷方式。在前面使用Vagrant创建的虚拟机中，我们将使用Docker安装本书所需的数据库、Web服务器和其他应用程序。实际上，前面创建Vagrant VM时，已经演示了如何使用Docker安装并运行MongoDB实例。

下面重启这个VM（因为前面使用命令vagrant halt停止了它）和MongoDB：

```
$> vagrant up
$> vagrant ssh
 vagrant@vagrant-ubuntu-trusty-64:~$ docker start mongoDB
 vagrant@vagrant-ubuntu-trusty-64:~$ docker ps
CONTAINER ID        IMAGE           COMMAND                 CREATED
360f5340d5fc        mongo:2         "/entrypoint.sh mong    41 minutes ago
```

```
STATUS              PORTS                     NAMES
Up 41 minutes       0.0.0.0:27017->27017/tcp  mongoDB
vagrant@vagrant-ubuntu-trusty-64:~$ exit
```

我们使用命令`docker start`启动容器，再使用命令`docker ps`列出当前运行的所有进程。

通过这个流程，可在眨眼之间重建一个全栈环境。你可能想知道，这真的像看起来那么神奇吗？答案是肯定的。Vagrant和Docker让开发人员能够专注于该做的事情，不用操心复杂的安装和棘手的配置。另外，我们做了额外的工作，以提供所有必要的步骤和资源，让你能够重建并测试本书所有代码示例和演示程序。

2.3　构建工具

随着时间的推移，代码的规模和复杂程度通常呈增长趋势。这是软件行业的性质决定的：所有产品都在生命周期内不断演进——实现客户提出的新需求。构建工具最大限度简化了项目生命周期的管理工作，它要求你遵循一些编码约定，如以特定方式组织代码、采用特定的类命名约定、将各种文件夹和文件组织成特定的项目结构。

有些读者可能熟悉Maven或Ant，它们都是项目处理方面的“瑞士军刀”，但本书的主要目标是介绍TDD，因此我们决定使用Gradle。Gradle的优点之一是样板式代码较少，因此其配置文件更简短、更易于理解。Gradle还是Google使用的构建工具之一。它得到IntelliJ IDEA的支持，学习和使用都非常容易，且大部分功能和任务都是通过插件提供的。

> 精通Gradle并非本书的目标，如果你要更深入地学习这款出色的工具，请访问其官网（http://gradle.org/），了解可使用的插件和可定制的选项。有关各种Java构建工具的比较，请参阅http://technologyconversations.com/2014/06/18/build-tools/。
>
> 接着往下阅读前，请确认你的系统安装了Gradle。

下面分析一个`build.gradle`文件中相关的部分。这种文件以简洁的方式存储了项目信息，使用的描述符语言为Groovy。下面是我们项目的构建文件，这是由IntelliJ自动生成的：

```
apply plugin: 'java'
sourceCompatibility = 1.7
version = '1.0'
```

由于这是一个Java项目，因此应用了一个Java插件，这让你能够执行常见的Java任务，如构建、打包、测试等。源代码兼容性被设置为JDK 7，因此如果你使用了这个版本不支持的Java语言，编译器将提出抗议。

```
repositories {
    mavenCentral()
}
```

Maven Central（http://search.maven.org/）存储了我们项目的所有依赖项，这部分告诉Gradle去哪里获取这些依赖项。就这个项目而言，Maven Central仓库足够了；但如果需要，你也可添加自定义仓库，如Nexus和ivy。

```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

最后，这部分演示了如何声明项目的依赖项——IntelliJ决定使用测试框架JUnit。

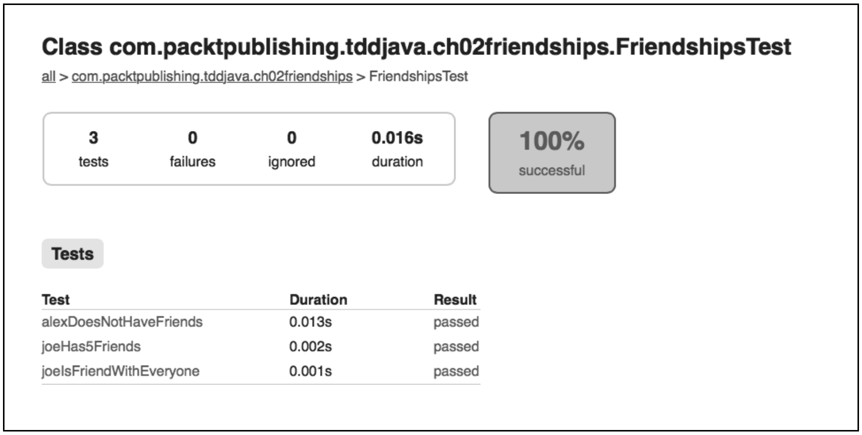
运行Gradle任务很容易，例如，要从命令行运行测试，只需执行如下命令：

```
gradle test
```

这项工作也可在IDEA中完成：选择菜单**View > Tool Windows > Gradle**打开Gradle Tool Window，再运行其中的测试任务。

测试结果存储在目录**build /reports/tests**下的HTML文件中。

下面是对示例代码执行命令`gradle test`生成的测试报告。



2.4　集成开发环境

鉴于本书介绍的工具和技术很多，我们推荐使用IntelliJ IDEA开发代码，主要是因为这个IDE不要求做繁琐的配置。其社区版（**IntelliJ IDEA CE**）自带了大量内置功能和插件，让你能够轻松而高效地编写代码。它根据扩展名自动推荐可安装的插件。由于本书使用的是IntelliJ IDEA，因此介绍相关步骤时，说的都是在IntelliJ IDEA中如何做；如果你使用的是其他IDE，则应采取相应的方式执行这些步骤。有关如何下载并安装IntelliJ IDEA，请参阅https://www.jetbrains.com/idea/。

IDEA 演示项目

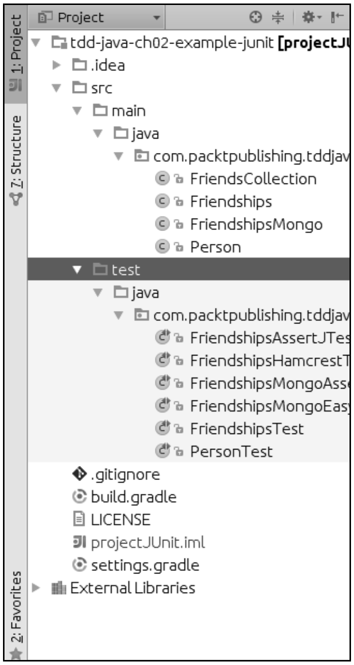
下面创建演示项目的基本布局。本章始终都将通过这个项目演示涉及的各种主题。这个项目将Java用作编程语言，并使用Gradle（http://gradle.org/）执行各种任务，如构建、测试等。

下面将包含本章示例的仓库导入IDEA。

(1) 启动IntelliJ IDEA，选择**Check out from Version Control**并单击**Git**。

(2) 在文本框Git repository URL中输入https://bitbucket.org/vfarcic/tdd-java-ch02-example-junit.git，单击**Clone**。不断**确认**IDEA提出的问题，直到使用从前述Git仓库克隆的代码新建一个项目。

导入的项目类似下图。



项目创建好后，该看看单元测试框架了。

2.5　单元测试框架

本节简要介绍两个最常用的Java单元测试框架——**JUnit**和**TestNG**，重点是通过比较使用它们编写的测试类阐述二者语法和主要功能。虽然存在细微的差别——主要是执行和组织测试的方式，但这两个框架都提供了最常用的功能。

我们从一个问题着手：测试是什么？如何定义？

> 测试是一个可重复的过程或方法，用于验证受测对象在指定环境下的行为是否正确，即向它提供指定的输入，并期望出现预定的输出或交互。

编程领域有多种范围不同的测试：功能测试、验收测试和单元测试，后面将更深入地探索这些测试类型。

单元测试旨在对一小块代码进行测试。下面看看如何测试一个Java类，这个类很简单，但足以激发你的兴趣：

```
public class Friendships {
    private final Map<String, List<String>> friendships =
      new HashMap<>();

    public void makeFriends(String person1, String person2) {
        addFriend(person1, person2);
        addFriend(person2, person1);
    }

    public List<String> getFriendsList(String person) {
        if (!friendships.containsKey(person)) {
            return Collections.emptyList();
        }
        return friendships.get(person);
    }

    public boolean areFriends(String person1, String person2) {
        return friendships.containsKey(person1) &&
        friendships.get(person1).contains(person2);
    }

    private void addFriend(String person, String friend) {
        if (!friendships.containsKey(person)) {
            friendships.put(person, new ArrayList<String>());
        }
        List<String> friends = friendships.get(person);
        if (!friends.contains(friend)) {
            friends.add(friend);
        }
    }
}
```

2.5.1　JUnit

JUnit（http://junit.org/）是一个用于编写和运行测试的框架，简单易学。每个测试都是一个方法，包含特定场景下将执行的部分代码。比较预期输出（行为）和实际输出（行为），以实现

代码验证。

下面是使用JUnit编写的测试类，其中并非涵盖所有场景，但这里要做的是让你知道测试长什么样。更佳的测试方式和最佳实践将在后面介绍。

测试类通常包含三个阶段：准备、测试和清理。下面先来看看为测试准备数据的方法。准备工作可在类层面执行，也可在方法层面执行：

```
Friendships friendships;

@BeforeClass
public static void beforeClass() {
  // 这个方法仅在初始化阶段执行一次
}

@Before
public void before() {
    friendships = new Friendships();
    friendships.makeFriends("Joe", "Audrey");
    friendships.makeFriends("Joe", "Peter");
    friendships.makeFriends("Joe", "Michael");
    friendships.makeFriends("Joe", "Britney");
    friendships.makeFriends("Joe", "Paul");
}
```

注解`@BeforeClass`指定，方法只在执行类中的测试方法前执行一次，非常适合用于执行大部分乃至全部测试都要求的一般性准备工作。

注解`@Before`指定，方法将在每个测试方法前运行，可使用它准备测试数据，这样就不用担心后面运行的测试修改数据状态。前面的示例中，我们实例化`Friendships`类，并在这个对象的列表中添加5个元素。不管各个测试将做什么样的修改，都将反复重建这些数据，直到所有测试都执行完毕。

这两种注解常用于准备数据库数据、创建测试所需的文件等。本书后面将介绍如何使用模拟对象消除外部依赖，但功能测试和集成测试可能需要外部依赖项，而注解`@Before`和`@BeforeClass`非常适合用于创建它们。

准备好数据后，执行实际测试：

```
@Test
public void alexDoesNotHaveFriends() {
  Assert.assertTrue("Alex does not have friends",
    friendships.getFriendsList("Alex").isEmpty());
}

@Test
public void joeHas5Friends() {
  Assert.assertEquals("Joe has 5 friends", 5,
```

```
        friendships.getFriendsList("Joe").size());
    }

    @Test
    public void joeIsFriendWithEveryone() {
      List<String> friendsOfJoe =
      Arrays.asList("Audrey", "Peter", "Michael", "Britney", "Paul");
    Assert.assertTrue(friendships.getFriendsList("Joe")
    .containsAll(friendsOfJoe));
    }
```


这个示例中，我们使用了众多断言中的几个。我们确认Alex确实没朋友，而Joe很讨人喜欢，有5个朋友（Audrey、Peter、Michael、Britney和Paul）。

最后，测试结束后，可能需要做些清理工作：

```
    @AfterClass
    public static void afterClass() {
      // 这个方法仅在所有测试都执行完毕后执行一次
    }

    @After
    public void after() {
       // 这个方法在每个测试执行完毕后都执行
    }
```

我们的示例（`Friendships`类）中，无需做任何清理工作，但有这种需求的情况下，这两种注解可提供所需的功能。它们的工作原理类似于注解`@Before`和`@BeforeClass`。`@AfterClass`指定的方法在所有测试都结束后运行一次，而注解`@After`指定的方法在每个测试结束后都执行。前面的示例中，每个测试方法都是在独立的类实例中执行的。因此只要没有使用全局变量和外部资源（如数据库和API），这些测试都将是彼此隔离的，即不管一个测试做什么，都不会影响其他测试。

完整的源代码可在FriendshipsTest类中找到，这个类包含在仓库https://github.com/TechnologyConversations/tdd-java-ch02-example-junit.git 和 https://bitbucket.org/vfarcic/tdd-java-ch02-example-junit.git中。

2.5.2 TestNG

TestNG（http://testng.org/doc/index.html）中，测试被组织成类，这与JUnit中完全相同。

要运行TestNG测试，必须添加下面的Gradle配置（`build.gradle`）：

```
dependencies {
   testCompile group: 'org.testng', name: 'testng', version: '6.8.21'
}

test.useTestNG(){
```

```
// 你可使用exclude/include过滤器指定要执行哪些测试
//excludeGroups 'complex'
}
```

不同于JUnit，要使用TestNG运行测试，必须添加额外的Gradle配置。

下面的测试类是使用TestNG编写的，其中包含的测试与前面使用JUnit所做的完全相同。这里省略了重复的导入和其他烦人的部分，旨在专注于相关部分：

```
@BeforeClass
public static void beforeClass() {
    // 这个方法仅在初始化阶段执行一次
}

@BeforeMethod
public void before() {
    friendships = new Friendships();
    friendships.makeFriends("Joe", "Audrey");
    friendships.makeFriends("Joe", "Peter");
    friendships.makeFriends("Joe", "Michael");
    friendships.makeFriends("Joe", "Britney");
    friendships.makeFriends("Joe", "Paul");
}
```

你可能注意到了JUnit和TestNG的相似之处。它们都使用注解指定方法的用途，且除注解名不同外（`@Before`和`@BeforeMethod`），没有其他不同。然而，不同于JUnit，TestNG使用同一个测试类实例执行所有测试方法。这意味着测试方法默认不是彼此隔离的，因此编写测试前后执行的方法时需要更加小心。

断言也很像：

```
public void alexDoesNotHaveFriends() {
  Assert.assertTrue(friendships.getFriendsList("Alex").isEmpty(),
  "Alex does not have friends");
}
public void joeHas5Friends() {
  Asert.assertEquals(friendships.getFriendsList("Joe").size(),
    5, "Joe has 5 friends");
}
public void joeIsFriendWithEveryone() {
  List<String> friendsOfJoe =
  Arrays.asList("Audrey", "Peter", "Michael", "Britney", "Paul");
  Assert.assertTrue(friendships.getFriendsList("Joe")
 .containsAll(friendsOfJoe));
}
```

与使用JUnit相比，唯一明显的差别是断言变量的排列顺序。JUnit断言中，参数依次为可选的消息、期望的值和实际值；而TestNG断言中，依次为实际值、期望的值和可选的消息。除向断言方法传递参数时的顺序不同外，JUnit和TestNG几乎没有其他不同之处。

你可能注意到了，这里没有使用注解`@Test`。TestNG中，可在类层级设置这一点，将所有公有方法都转换为测试。

`@After`注解也很像。唯一明显的差别是，与JUnit注解`@After`对应的TestNG注解为`@AfterMethod`。

正如你看到的，JUnit和TestNG的语法很像。测试被组织成类，而验证是使用断言执行的。这并不意味着这两个框架不存在重大差别，本书后面将介绍其中的一些。建议你自己探索JUnit（http://junit.org/）和TestNG（http://testng.org/）。

前述示例的完整源代码可在https://bitbucket.org/vfarcic/tdd-java-ch02-example-testng.git找到。

前面编写所有断言时，都只使用了测试框架，但有些测试工具可帮助我们编写更漂亮、更易于理解的断言。

2.6 Hamcrest 和 AssertJ

前一节中，我们概述了单元测试的定义及如何使用两种最常用的Java框架进行编写。测试是项目的重要部分，为何不改进其编写方式呢？一些出色的项目应运而生，旨在通过修改做出断言的方式强化测试的语义，让测试更简洁、更易于理解。

2.6.1 Hamcrest

Hamcrest添加了大量被称为“匹配器”的方法，其中每个匹配器都设计用于执行特定的比较操作。Hamcrest的可扩展性很好，让你能够创建自定义匹配器。另外，JUnit发布版包含Hamcrest的核心，提供了对Hamcrest的原生支持，这让你能够直接使用Hamcrest。但我们要使用功能齐备的Hamcrest，因此需要在Gradle配置文件中添加如下测试依赖项：

```
testCompile 'org.hamcrest:hamcrest-all:1.3'
```

下面比较等价的JUnit断言和Hamcrest断言：

- ❑ JUnit断言：

```
    List<String> friendsOfJoe =
Arrays.asList("Audrey", "Peter", "Michael", "Britney", "Paul");
    Assert.assertTrue( friendships.getFriendsList ("Joe")
   .containsAll(friendsOfJoe)
    );
```

- ❑ Hamcrest断言：

```
assertThat(
        friendships.getFriendsList("Joe"),
```

```
                containsInAnyOrder("Audrey", "Peter",
                 "Michael", "Britney", "Paul")
        );
```

正如你看到的，Hamcrest的表达力强些。它提供的断言多得多，让我们能够避免一些样板式代码，同时让代码更容易理解、更具表达力。

再来看一个例子：

- JUnit断言：

```
Assert.assertEquals(5, friendships.getFriendsList("Joe").size());
```

- Hamcrest断言：

```
assertThat(friendships.getFriendsList("Joe"), hasSize(5));
```

你应该注意到了两个不同之处。首先，不同于JUnit，Hamcrest几乎总是直接使用对象；而JUnit中，需要先获取整数表示的长度，再将其与期望的值（`5`）进行比较。Hamcrest提供的断言更多，让我们可将其中的一个（`hasSize`）与实际对象（`List`）结合使用。另一个不同之处是，Hamcrest像TestNG一样反转了参数的排列顺序，不将实际值作为第一个参数。

这两个示例不足以展现Hamcrest的全部潜力，本书后面将提供更多示例，并对Hamcrest做更详细的介绍。要更深入地了解其语法，请访问https://code.google.com/p/hamcrest/或http://hamcrest.org/。

完整的源代码可在`FriendshipsHamcrestTest`类中找到，这个类包含在仓库https://github.com/TechnologyConversations/tdd-java-ch02-example-junit.git 和 https://bitbucket.org/vfarcic/tdd-java-ch02-example-junit.git中。

2.6.2　AssertJ

AssertJ的工作原理与Hamcrest类似，一个重要的差别是AssertJ断言是可以串接的。

要使用AssertJ，必须在Gradle配置文件的dependencies部分添加如下依赖项：

```
testCompile 'org.assertj:assertj-core:2.0.0'
```

下面比较JUnit断言和AssertJ断言：

```
Assert.assertEquals(5, friendships.getFriendsList("Joe").size());
List<String> friendsOfJoe =
  Arrays.asList("Audrey", "Peter", "Michael", "Britney", "Paul");
Assert.assertTrue(  friendships.getFriendsList("Joe")
  .containsAll (friendsOfJoe)
);
```

AssertJ中，可将这两个断言串接成一个：

```
assertThat(friendships.getFriendsList("Joe"))
        .hasSize(5)
        .containsOnly("Audrey", "Peter", "Michael", "Britney",
           "Paul");
```

这是很大的改进：不需要两个独立的断言，也无需使用期望的值新建一个列表。另外，AssertJ断言的可读性更强，也更容易理解。

完整的源代码可在FriendshipsAssertJTest类中找到，这个类包含在仓库https://github.com/TechnologyConversations/tdd-java-ch02-example- junit.git和https://bitbucket.org/vfarcic/tdd-java-ch02-example-junit.git中。

创建并运行测试后，你可能想知道这些测试的代码覆盖率有多高。

2.7　代码覆盖率工具

编写了测试并不意味着它们很好，也不意味着它们的代码覆盖率足够高。开始编写并运行测试后，一种自然而然的反应是开始提出以前不会问的问题。正确地测试了代码的哪些部分？测试未考虑哪些情形？测试够详尽吗？要回答这些问题及其他类似问题，可使用代码覆盖率工具。这些工具可用于找出未被测试覆盖的代码块或代码行，它们还能够计算被覆盖的代码所占的百分比，以及提供其他有趣的指标。

使用这些功能强大的工具可获得相关的指标，找出测试代码并实现代码之间的关系。然而，与其他工具一样，这里有必要澄清其用途。它们并不能提供质量方面的信息，而只能告诉你哪些代码经过了测试。

> 代码覆盖率工具能够指出测试执行期间触及了哪些代码行，但并不能保证你遵循了良好的测试实践，因为这些指标中不包含测试质量。

下面介绍一个最受欢迎的代码覆盖率计算工具。

JaCoCo

Java Code Coverage（**JaCoCo**）是一个著名的测试覆盖率测量工具。要在我们的项目中使用它，需要在Gradle配置文件（build.gradle）中添加几行内容：

(1) 添加JaCoCo插件：

```
apply plugin: 'jacoco'
```

(2) 为查看JaCoCo的结果，从命令提示符运行如下命令：

```
gradle test jacocoTestReport
```

(3) 也可通过**IDEA Tool Window**执行这个任务。

(4) 最终的结果存储在目录**build/reports/jacoco/test/html**中，这是一个HTML文件，可使用任何浏览器打开。

Friendships

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
areFriends(String, String)		0%		0%	3	3	1	1	1	1
addFriend(String, String)		100%		75%	1	3	0	4	0	1
getFriendsList(String)		100%		100%	0	2	0	2	0	1
makeFriends(String, String)		100%		n/a	0	1	0	3	0	1
Friendships()		100%		n/a	0	1	0	2	0	1
Total	17 of 75	77%	5 of 10	50%	4	10	1	12	1	5

本书后面将更详细地介绍JaCoCo，届时可访问http://www.eclemma.org/jacoco/获取更详细的信息。

2.8　模拟框架

我们的项目看起来很不错，但太简单，与真实的项目相差很远——它没有使用外部资源。而Java项目大多都需要使用数据库，下面就来引入数据库。

对于使用外部资源或第三方库的代码，通常如何测试呢？答案是使用模拟对象。模拟对象是可用于替代实际对象的仿真对象，在依赖的外部资源不可用时很有用。

事实上，开发应用程序期间，根本不需要数据库。你可使用模拟对象提高开发和测试的速度，仅在进入运行阶段后才使用真正的数据库连接。你不用花时间建立数据库和准备测试数据，而是专注于编写类，等到集成阶段再考虑这些问题。

为方便演示，我们将引入两个新类——`Person`和`FriendCollection`。持久化工作将使用MongoDB（https://www.mongodb. org/）完成。

`Person`类表示数据库对象数据，而`FriendCollection`将充当数据访问层。但愿这些类的代码是不言自明的。

下面创建`Person`类：

```
public class Person {
    @Id
    private String name;

    private List<String> friends;

    public Person() { }
```

```
    public Person(String name) {
        this.name = name;
        friends = new ArrayList<>();
    }

    public List<String> getFriends() {
        return friends;
    }

    public void addFriend(String friend) {
        if (!friends.contains(friend)) friends.add(friend);
    }

}
```

接下来创建`FriendsCollection`类：

```
public class FriendsCollection {
    private MongoCollection friends;

    public FriendsCollection() {
        try {
            DB db = new MongoClient().getDB("friendships");
            friends = new Jongo(db).getCollection("friends");
        } catch (UnknownHostException e) {
            throw new RuntimeException(e.getMessage());
        }
    }

    public Person findByName(String name) {
        return friends.findOne("{_id: #}", name).as(Person.class);
    }

    public void save(Person p) {
        friends.save(p);
    }
}
```

另外，引入了一些新的依赖项，因此需要修改Gradle配置文件的dependencies部分。第一个依赖项是**MongoDB**驱动程序，用于连接到数据库；第二个是**Jongo**，能够轻松访问MongoDB集合的小型项目。

在Gradle配置文件中添加依赖项`MongoDB`和`Jongo`的代码如下所示：

```
dependencies {
    compile 'org.mongodb:mongo-java-driver:2.13.2'
    compile 'org.jongo:jongo:1.1'
}
```

我们现在使用了数据库，因此必须修改`Friendships`类。为此，将其中的映射改为`FriendsCollection`，并修改使用它的其他代码，最终结果如下所示：

```
public class FriendshipsMongo {
    private FriendsCollection friends;

    public FriendshipsMongo() {
        friends = new FriendsCollection();
    }

    public List<String> getFriendsList(String person) {
        Person p = friends.findByName(person);
        if (p == null)  return Collections.emptyList();
        return p.getFriends();
    }

    public void makeFriends(String person1, String person2) {
        addFriend(person1, person2);
        addFriend(person2, person1);
    }

    public boolean areFriends(String person1, String person2) {
        Person p = friends.findByName(person1);
        return p != null && p.getFriends().contains(person2);
    }

    private void addFriend(String person, String friend) {
        Person p = friends.findByName(person);
        if (p == null)  p  = new Person(person);
        p.addFriend(friend);
        friends.save(p);
    }
}
```

完整的源代码可在`FriendsCollection`和`FriendshipsMongo`类中找到，这些类包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-junit.git中。

将`Friendships`类改为使用MongoDB的`FriendshipsMongo`类后，下面看一种使用模拟对象对其进行测试的方式。

2.8.1　Mockito

Mockito这个Java框架让你能够轻松创建测试替身。要使用它，需要在Gradle配置文件中添加如下依赖项：

```
dependencies {
  testCompile group: 'org.mockito', name: 'mockito-all', version: '1.+'
}
```

Mockito是通过JUnit运行器运行的，它替我们创建所有必需的模拟对象，并将其注入包含测试的类。有两种基本方法：手工实例化模拟对象，并通过类构造函数将它们作为类依赖项注入；使用一系列注解。下面的示例中，我们将演示如何使用注解注入模拟对象。

要让一个类能够使用Mockito注解，必须使用MockitoJUnitRunner运行。使用这个运行器可简化工作，因为你只需给要创建的对象添加注解：

```
@RunWith(MockitoJUnitRunner.class)
public class FriendshipsTest {
...
}
```

测试类中，需要使用@InjectMocks标注受测类，以告诉Mockito应将模拟对象注入哪个类：

```
@InjectMocks
FriendshipsMongo friendships;
```

接下来，需要指定要将这个类（FriendshipsMongo）中的哪些方法或对象替换为模拟对象：

```
@Mock
FriendsCollection friends;
```

这个示例中，将模拟FriendshipsMongo类中的FriendsCollection。

现在，可以指定friends被调用时应返回的值：

```
Person joe = new Person("Joe");
doReturn(joe).when(friends).findByName("Joe");
assertThat(friends.findByName("Joe")).isEqualTo(joe);
```

这个示例中，我们让Mockito在friends.findByName("joe")被调用时返回对象joe；然后，使用assertThat验证这种假设是正确的。

下面再来做一个测试，这个测试与前面未使用MongoDB时所做的相同：

```
@Test
public void joeHas5Friends() {
    List<String> expected =
Arrays.asList("Audrey", "Peter", "Michael", "Britney", "Paul");
    Person joe = spy(new Person("Joe"));

    doReturn(joe).when(friends).findByName("Joe");
    doReturn(expected).when(joe).getFriends();

    assertThat(friendships.getFriendsList("Joe"))
      .hasSize(5)
 .containsOnly("Audrey", "Peter", "Michael", "Britney", "Paul");
}
```

这个小小的测试中，发生的事情很多。首先，我们将joe指定为间谍（spy）。Mockito中，除非另有说明，否则间谍是使用真实方法的真实对象。接下来，我们让Mockito在friends.findByName("joe")被调用时都返回对象joe，并在joe.getFriends()被调用时返回列表expected，这样getFriendsList被调用时将返回列表expected。最后，我们使用断言确认getFriendsList确实返回了姓名列表expected。

完整的源代码可在FriendshipsMongoAssertJTest类中找到，这个类包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-junit.git中。

本书后面将使用Mockito，届时你将有机会更深入了解Mockito和模拟技术。有关Mockito的更详细信息请参阅http://mockito.org/。

2.8.2　EasyMock

EasyMock是另一种模拟框架，它与Mockito很像，主要差别在于EasyMock创建的不是间谍对象，而是模拟对象，其他差别都属于语法方面。

下面看一个EasyMock示例，它使用的测试用例与Mockito示例相同：

```
@RunWith(EasyMockRunner.class)
public class FriendshipsTest {
    @TestSubject
    FriendshipsMongo friendships = new FriendshipsMongo();
    @Mock(type = MockType.NICE)
    FriendsCollection friends;
```

从本质上说，EasyMock运行器的作用与Mockito运行器相同。

```
@TestSubject
FriendshipsMongo friendships = new FriendshipsMongo();

@Mock(type = MockType.NICE)
FriendsCollection friends;
```

注解`@TestSubject`类似于Mockito注解`@InjectMocks`，而注解`@Mock`类似于Mockito注解`@Mock`，也标注要模拟的对象。另外，type值`NICE`让模拟对象返回空（null）。

下面比较前面使用Mockito做过的一个断言：

```
    @Test
    public void mockingWorksAsExpected() {
        Person joe = new Person("Joe");
        expect(friends.findByName("Joe")).andReturn(joe);
        replay(friends);
        assertThat(friends.findByName("Joe")).isEqualTo(joe);
    }
```

除细微的语法差别外，EasyMock的唯一劣势是需要添加额外的指令`replay`，让前面指定的期望生效。其他代码几乎完全相同。我们指定`friends.findByName`应返回对象`joe`，让这个期望生效，再使用断言检查实际结果是否符合预期。

我们使用Mockito编写的第二个测试方法的EasyMock版本如下：

```
    @Test
    public void joeHas5Friends() {
```

```
    List<String> expected =
  Arrays.asList("Audrey", "Peter", "Michael", "Britney", "Paul");
    Person joe = createMock(Person.class);

    expect(friends.findByName("Joe")).andReturn(joe);
    expect(joe.getFriends()).andReturn(expected);
    replay(friends);
    replay(joe);

    assertThat(friendships.getFriendsList("Joe"))
            .hasSize(5)
            .containsOnly("Audrey", "Peter", "Michael", "Britney",
              "Paul");
}
```


同样，相比于Mockito版本几乎没什么不同，只是EasyMock没有间谍。在有些情况下，这可能是重大的差别。

虽然这两个框架很像，但考虑到一些细节，我们决定在本书中始终使用Mockito。

> 有关模拟框架EasyMock的更详细信息，请参阅http://easymock.org/。

完整的源代码可在FriendshipsMongoEasyMockTest类中找到，这个类包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-junit.git和https://github.com/TechnologyConversations/tdd-java-ch02-example-junit.git中。

2.8.3 PowerMock

前面介绍的两个框架并未涵盖所有类型的方法和字段。

对于有些类、方法或字段，Mockito和EasyMock可能无法提供模拟支持，这取决于指定的限定符（如static或final）。这种情况下，可使用PowerMock扩展模拟框架，这样就能模拟很棘手的对象。然而，务必要慎用PowerMock，因为如果必须使用它提供的很多功能，通常昭示着设计很糟糕。处理遗留代码时，PowerMock可能是不错的选择；其他情况下，应尽量以合理的方式设计代码，确保不需要使用PowerMock，本书后面将介绍。

有关PowerMock的更详细信息，请参阅https://code.google.com/p/powermock/。

2.9 用户界面测试

虽然单元测试能够也应该覆盖应用程序的很大一部分，但依然需要功能测试和验收测试。不同于单元测试，它们提供更高层面的验证，通常在入口处执行，且严重依赖用户界面。归根结底，大部分情况下，我们创建的应用程序是供人类使用的，因此确信应用程序行为正常至关重要。要

获得这样的信心，需要站在最终用户的角度对应用程序进行测试，确保其行为符合预期。

下面概述如何通过用户界面进行功能测试和验收测试。我们将以Web为例，虽然有很多其他类型的用户界面，如桌面应用程序、智能手机界面等。

2.9.1　Web 测试框架

本章前面测试了应用程序类和数据源，但还遗漏了一项，那就是最常见的用户入口——Web。大多数企业应用（如内联网和公司网站）都是通过浏览器访问的，所以Web测试意义重大，可帮助我们确信其行为符合预期。

另外，每当应用发生变化时，公司都投入大量时间执行漫长而繁重的手工测试。这是巨大的浪费，因为通过使用诸如**Selenium**和**Selenide**等工具，很多测试都可自动化，在无人值守的情况下执行。

2.9.2　Selenium

Selenium是一款出色的Web测试工具，它使用浏览器运行验证，并支持所有流行的浏览器，如Firefox、Safari和Chrome。它还支持使用无界面浏览器（headless browser）测试网页，这样可极大地提高速度，同时减少资源消耗。

一款名为SeleniumIDE的插件可通过记录用户执行的操作创建测试，但当前只有Firefox支持。遗憾的是，以这种方式生成的测试虽然能快速提供结果，但它们通常极其脆弱，最终必然带来问题，尤其是网页的某些部分发生变化时。有鉴于此，我们将坚持在编写测试代码时始终不求助于这个插件。

要执行Selenium，最简单的方法是通过JUnitRunner运行。所有Selenium测试都首先初始化`WebDriver`——用于同浏览器通信的类：

(1) 我们首先在Gradle配置文件中添加依赖项：

```
dependencies {
  testCompile 'org.seleniumhq.selenium:selenium- java:2.45.0'
}
```

(2) 作为示例，我们将创建一个在维基百科中搜索的测试，并使用Firefox驱动程序：

```
WebDriver driver = new FirefoxDriver();
```

WebDriver是一个接口，可使用Selenium提供的众多驱动程序之一进行实例化：

(1) 使用如下指令打开一个URL：

```
driver.get
  ("http://en.wikipedia.org/wiki/Main_Page");
```

(2) 网页打开后，可根据名称找到搜索框并指定要搜索的内容：

```
WebElement query =
driver.findElement(By.name("search"));
query.sendKeys("Test-driven development");
```

(3) 指定要搜索的内容后，找到并单击**Go**按钮：

```
WebElement goButton =
driver.findElement(By.name("go"));
goButton.click();
```

(4) 进入目标网页后进行验证，这里是为了确定页面标题正确无误：

```
assertThat(driver.getTitle(),
startsWith("Test-driven development"));
```

(5) 使用完驱动程序后，应将其关闭：

```
driver.quit();
```

就这么简单。我们创建了一个很小但很有用的测试，它验证单个用例。对于Selenium，虽然可说的还有很多，但前面的介绍应足以让你认识到其潜力。

> 有关Selenium的更详细信息以及WebDriver的更复杂用法，请参阅http://www.seleniumhq.org/。

完整的源代码可在SeleniumTest类中找到，这个类包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-web.git中。

Selenium是最常用的Web测试框架，但它很低级，需要做大量微调。如果有一个更高级的库，能够实现一些常见模式并解决反复出现的需求，Selenium将有用得多。Selenide正是基于这种理念开发的。

2.9.3 Selenide

从前面的介绍可知，Selenium很酷，让我们能够核实应用程序是否表现优异。但有些情况下，配置和使用起来有点棘手。Selenide是一个基于Selenium的项目，提供了优良的测试编写语法，提高了测试的可读性。它将WebDriver和配置隐藏，同时提供了极大的定制空间：

(1) 与前面使用的其他库一样，首先需要在Gradle配置文件中添加依赖项：

```
dependencies {
    testCompile 'com.codeborne:selenide:2.17'
}
```

(2) 下面看看如何使用Selenide编写前面的Selenium测试。熟悉jQuery（https://jquery.com/）的读者对这里的语法可能不会感到陌生：

```
public class SelenideTest {
    @Test
    public void wikipediaSearchFeature() throws
      InterruptedException {
        // 打开维基百科页面
        open("http://en.wikipedia.org/wiki/Main_Page");

        // 搜索TDD
        $(By.name("search")).setValue("Test-driven" +
          "development");

        // 单击搜索按钮
        $(By.name("go")).click();

        // 检查结果
        assertThat(title(), startsWith("Test-driven" +
          "development"));
    }
}
```

这种测试编写方式的表达力更强，不但语法更流畅，这些代码后面也自动执行了一些操作——使用Selenium时需要编写额外的代码行执行。例如，单击操作将等到目标元素可用后再执行，且仅在等待时间超过指定时间时，这个操作才会失败。而Selenium中，如果目标元素不可用，单击操作将立即失败。当前，很多元素都是通过JavaScript动态加载的，不能指望所有元素都会立即出现。因此，Selenide提供的这项功能很有用，让我们无需反复编写样板式代码。Selenide还带来了众多其他好处。鉴于Selenide相比于Selenium存在这些优点，本书将始终使用它进行Web测试。另外，后面有一章专门介绍如何使用这个框架进行Web测试。有关如何在测试中使用Web驱动程序的更详细信息，请参阅http://selenide.org/。

不管测试是使用哪个框架编写的，效果都相同：测试运行时，将出现一个Firefox浏览器窗口，并依次执行测试指定的步骤。除非使用的是无界面浏览器，否则你将看到整个测试过程。如果发生错误，可使用故障跟踪（failure trace）。另外，你还可随时截取浏览器屏幕快照，例如，一种常见的做法是将出现故障时的情形记录下来。

完整的源代码可在SelenideTest类中找到，这个类包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-web.git中。

对Web测试框架有基本认识后，下面看看BDD。

2.10 行为驱动开发

行为驱动开发（**BDD**）是一种敏捷过程，旨在整个项目开发过程中都专注于相关方的利益。BDD基于这样的前提，即需求必须以所有人（业务代表、分析师、开发人员、测试人员、项目经理等）都能看懂的方式编写。这里的关键是提供一组人人都能理解并使用的独特工件——一系列用户故事。这些故事由整个团队编写，并被用作需求和可执行的测试用例。这是一种TDD实施方式，但具有单元测试无法比拟的清晰度；这还是一种描述和测试功能的方式，其中的测试几乎完全是使用自然语言编写的，但既是可运行的，又是可重复的。

故事由场景组成，而每个场景都是使用自然语言编写的，表示一个简洁的用例，由不同步骤组成。步骤按顺序排列，定义了场景的前置条件、事件和结果。每个步骤都以单词Given、When或Then打头，其中Given用于指定前置条件，When用于指定操作，而Then用于执行验证。

这里只是简要的介绍，本书后面有一章（第7章）专门介绍这个主题。下面介绍JBehave和Cucumber——众多故事编写和执行框架中的两个。

2.10.1 JBehave

JBehave是一个Java BDD框架，用于编写可执行和自动化的验收测试。故事中的步骤被关联到Java代码，这是通过使用这个框架提供的注解实现的：

(1) 首先，在Gradle配置文件中添加JBehave依赖项：

```
dependencies {
...
    testCompile 'org.jbehave:jbehave-core:3.9.5'
...
}
```

(2) 下面看几个步骤：

```
@Given("I go to Wikipedia homepage")
public void goToWikiPage() {
    open("http://en.wikipedia.org/wiki/Main_Page");
}
```

(3) 这是一个Given步骤，表示要成功执行后续操作必须满足的一个前置条件，此处指打开一个维基百科网页。指定前置条件后，定义一些操作：

```
@When("I enter the value $value on a field named " +
  "$fieldName")
public void enterValueOnFieldByName(String value,
  String fieldName){
    $(By.name(fieldName)).setValue(value);
}
```

```
@When("I click the button $buttonName")
public void clickButonByName(String buttonName){
    $(By.name(buttonName)).click();
}
```

(4) 正如你看到的，操作是使用注解`@When`定义的。在这里，我们使用这些步骤设置一个文本框的值并单击特定按钮。操作执行完毕后进行验证，注意，引入参数可让步骤更灵活：

```
@Then("the page title contains $title")
public void pageTitleIs(String title) {
    assertThat(title(), containsString(title));
}
```

验证是使用注解`@Then`声明的。这个示例中，我们验证页面标题符合预期。

这些步骤可在`WebSteps`类中找到，这个类包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-web.git中。

定义并使用步骤。下面的故事将这些步骤组合在一起，旨在验证期望的行为：

```
Scenario: TDD search on wikipedia
```

它首先指定了场景名称。场景名的唯一目的是提供足够的信息，应尽可能简洁，同时能明确标识用例。

```
Given I go to Wikipedia homepage
When I enter the value Test-driven development on a field named search
When I click the button go
Then the page title contains Test-driven development
```

正如你看到的，这里使用了前面定义的步骤文本。依次执行与这些步骤相关联的代码，如果有代码出现问题，将终止执行，而场景本身将被视为失败的。

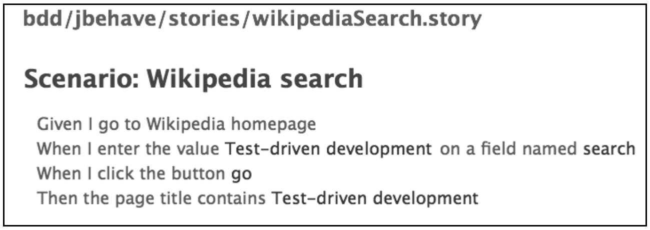
虽然这里是在故事前定义的步骤，但也可按相反的顺序做——先定义故事再定义步骤。这种情况下，场景将处于**悬置**（**pending**）状态，这意味着缺失必要的步骤。

这个故事可在文件`wikipediaSearch.story`中找到，这个文件包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-web.git中。

为运行这个故事，执行如下命令：

```
$> gradle testJBehave
```

这个故事运行时，你将在浏览器中看到执行的操作。运行完毕后，生成一个包含执行结果的报告，它存储在目录`build/reports/jbehave`中。



JBehave故事执行报告

为简洁起见，这里删除了运行JBehave故事的`build.gradle`代码。完整的源代码可在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-web.git 中找到。

> 有关JBehave的更详细信息及其带来的好处，请参阅http://jbehave.org/。

2.10.2 Cucumber

Cucumber最初是一个Ruby BDD框架，但现在支持包括Java在内的多种语言，它提供的功能与JBehave很像。

下面看看如何使用Cucumber编写前面的示例。

与之前使用的其他框架一样，要使用Cucumber，也必须在配置文件`build.gradle`中添加相应的依赖项：

```
dependencies {
    ...
    testCompile 'info.cukes:cucumber-java:1.2.2'
    testCompile 'info.cukes:cucumber-junit:1.2.2'
    ...
}
```

使用Cucumber创建前面使用JBehave时创建的步骤：

```
@Given("^I go to Wikipedia homepage$")
public void goToWikiPage() {
    open("http://en.wikipedia.org/wiki/Main_Page");
}

@When("^I enter the value (.*) on a field named (.*)$")
public void enterValueOnFieldByName(String value,
    String fieldName){
    $(By.name(fieldName)).setValue(value);
}

@When("^I click the button (.*)$")
public void clickButonByName(String buttonName){
```

```
        $(By.name(buttonName)).click();
    }

    @Then("^the page title contains (.*)$")
    public void pageTitleIs(String title) {
        assertThat(title(), containsString(title));
    }
```

这两个框架唯一显著的差别在于Cucumber定义步骤文本的方式，它使用正则表达式匹配变量类型，而不像JBehave那样根据方法签名推断。

这些步骤的代码可在`WebSteps`类中找到，而这个类包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-web.git中。

下面看看使用Cucumber语法编写的故事：

```
Feature: Wikipedia Search

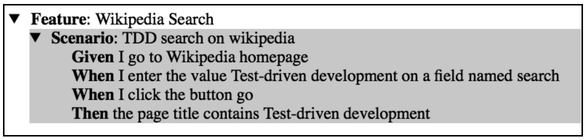
  Scenario: TDD search on wikipedia
    Given I go to Wikipedia homepage
    When I enter the value Test-driven development on a field named search
    When I click the button go
    Then the page title contains Test-driven development
```

几乎没什么不同。这个故事可在文件`wikipediaSearch.feature`中找到，这个文件包含在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-web.git中。

你可能猜到了，要运行Cucumber故事，只需执行如下Gradle任务：

```
$> gradle testCucumber
```

结果报告存储在目录`build/reports/cucumber-report`中。上述故事的报告如下：



Cucumber故事的执行报告

完整的代码示例可在仓库https://bitbucket.org/vfarcic/tdd-java-ch02-example-web.git中找到。

> 有关Cucumber支持的语言清单和其他细节，请参阅https://cukes.info/。

鉴于JBehave和Cucumber提供的功能类似，我们决定在本书后面都使用JBehave。本书后面有一章专门介绍BDD和JBehave。

2.11　小结

本章暂时放下TDD，转而介绍了本书后面演示代码时需要用到的众多工具和框架——从版本控制、虚拟机、构建工具和IDE到当前常用的测试框架。

我们是开源运动的坚定支持者，本着这种精神，对于每类工具和框架，我们都尽力选择免费的。

准备好需要的所有工具后，下面更深入地探索TDD——从TDD的中流砥柱“红灯-绿灯-重构”着手。

第 3 章

红灯–绿灯–重构——从失败到成功再到完美

“光知道还不够，还必须付诸应用；光有决心还不够，还必须行动。”

——李小龙

“红灯–绿灯–重构”流程是TDD的基石，这个过程就像玩乒乓球，以极快的速度在测试和实现代码之间切换。期间将失败，然后成功，最后改进。

本章将开发一个“井字游戏”，期间每次只考虑一个需求。我们首先编写一个测试，看看它是否未通过；然后编写实现这个测试的代码，运行所有测试并看看它们是否都通过；最后，通过重构改进代码。这个过程将重复多次，直到成功实现所有需求。

我们将首先使用Gradle和JUnit搭建环境，然后更深入地介绍“红灯–绿灯–重构”过程。搭建好环境并做好理论上的准备工作后，我们将确定这个应用程序的粗略需求。

一切都准备就绪后，投入开发工作——每次解决一个需求。完成开发工作后，检查代码覆盖率，并据此判断结果可以接受还是需要添加更多测试。

本章涵盖如下主题：

- 使用Gradle和JUnit搭建环境；
- “红灯–绿灯–重构”过程；
- “井字游戏”的需求；
- 开发“井字游戏”；
- 代码覆盖率；
- 更多练习。

3.1　使用 Gradle 和 JUnit 搭建环境

你很可能熟悉如何创建Java项目，但可能没有使用过IntelliJ IDEA，或者使用的构建工具是Maven而不是Gradle。为确保你能按本书介绍的方法做，下面简要说明如何创建项目。

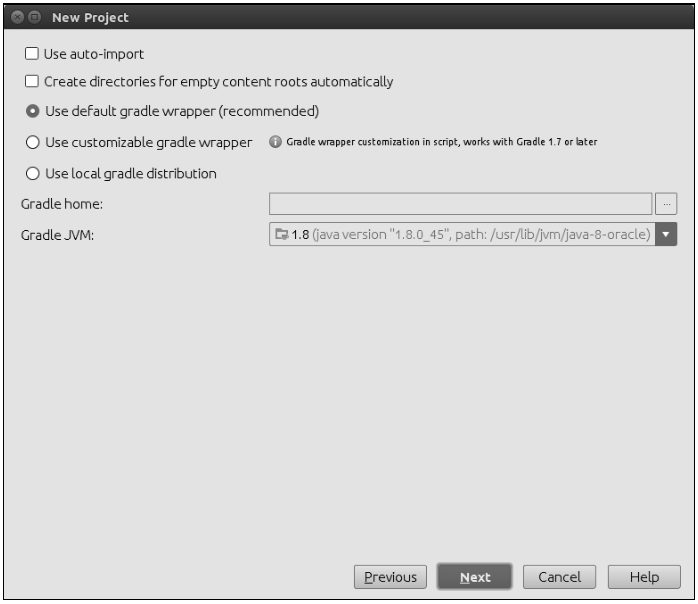
在 IntelliJ IDEA 中创建 Gradle/Java 项目

本书的主要目标是介绍TDD，因此不会详细介绍Gradle和IntelliJ IDEA。Gradle和IntelliJ IDEA都只是手段，本书的所有练习都可使用其他IDE和构建工具完成，例如，可使用Maven和Eclipse。从很大程度上说，完全按本书说的做可能更容易，但如何选择由你决定。

要在IntelliJ IDEA中新建一个Gradle项目，可按下面的步骤做：

(1) 启动**IntelliJ IDEA**，单击**Create New Project**并从左边的列表中选择**Gradle**，再单击**Next**按钮。

(2) 如果使用的是IDEA 14或更高版本，将要求指定工件（**Artifact**）**ID**。输入**tdd-java-ch03-tic-tac-toe**并单击**Next**按钮两次，再将项目名指定为**tdd-java-ch03-tic-tac-toe**，然后单击**Finish**按钮。



从**New Project**窗格可知，IDEA已经创建了文件`build.gradle`。打开这个文件，可以发现其中已经包含**JUnit**依赖项。本章只使用这个框架，因此不需要再添加其他配置。默认情况下，`build.gradle`将源代码兼容性设置指定为Java 1.5，但可将其修改为任何你喜欢的版本。本章示

例不会使用Java 1.5之后推出的新功能，但这并不意味着不能使用更高的版本（如JDK 8）完成这个练习。

这个项目的`build.gradle`文件应类似于下面这样：

```
apply plugin: 'java'

version = '1.0'

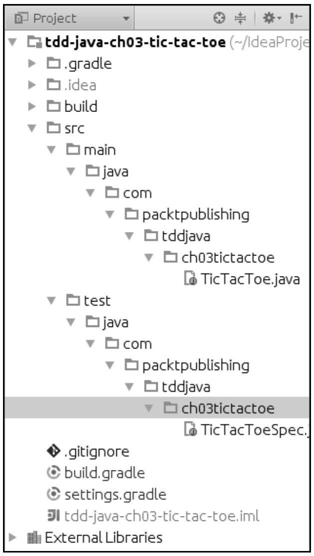
repositories {
    mavenCentral()`
}

dependencies {
    testCompile group: 'junit', name: 'junit',
        version: '4.11'
}
```

现在，余下的唯一工作是创建测试包和实现包。在**Project**窗格中右击打开**上下文**菜单，选择**New > Directory**，输入`src/test/java/com/packtpublishing/tddjava/ch03tictactoe`并单击**OK**按钮，以创建测试包。重复上述步骤，但输入目录`src/main/java/com/packtpublishing/tddjava/ch03tictactoe`，以创建实现包。

最后，我们需要创建测试类和实现类。为此，在目录`src/test/java`下的`com.packtpublishing.tddjava.ch03tictactoe`包中创建`TicTacToeSpec`类，它将包含所有测试。重复上述过程，在目录`src/main/java`中创建`TicTacToe`类。

此时，项目结构应类似于下图。



可在Git仓库`tdd-java-ch03-tic-tac-toe`的`00-setup`分支（https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/00-setup）中找到源代码。

> **务必将测试和实现代码分开。**
>
> 这样做的好处是：可避免不小心将测试与产品二进制文件一起打包；很多构建工具都假定测试位于特定的源代码目录。
>
> 一种常见的做法是，确保至少有两个源代码目录：将实现代码放在目录`src/main/java`中，并将测试代码放在目录`src/test/java`中。较大的项目中，源代码目录可能更多，但依然必须将实现和测试分开。
>
> 诸如Maven和Gradle等构建工具要求将源代码目录分开，还要求遵循特定的命名约定。

准备工作至此完成，可以开始开发“井字游戏”了。将JUnit用作测试框架，并使用Gradle执行编译、依赖、测试和其他任务。第1章简要介绍了“红灯–绿灯–重构”过程，它是TDD的中流砥柱，而本章练习的主要目标就是熟悉这个过程。因此，着手开发“井字游戏”前，需要对整个过程进行更详细的介绍。

3.2 “红灯–绿灯–重构”过程

“红灯–绿灯–重构”过程是TDD最重要的组成部分，是最主要的支柱。如果没有它，TDD的其他方面根本行不通。

这个名称源自代码在周期内的状态：处于红灯状态时，代码不管用；处于绿灯状态时，一切都像预期的那样工作，但并不一定是最佳的；到了重构阶段，我们知道测试很好地覆盖了各项功能，可以充满信心地去修改它，让它变得更好。

3.2.1 编写一个测试

每次添加新功能时都首先编写一个测试，这旨在编写代码前专注于需求和代码设计。测试是可执行的文档，以后能够帮助理解代码目的及其背后的意图。

当前，我们处于红灯状态，因为测试执行时以失败告终，即测试对代码的期望和代码实际的功能之间存在差距。更具体地说，没有代码满足最后一个测试的期望，因为我们还没有编写这样的代码。在这个阶段，可能所有测试都通过了，但这昭示着存在问题。

3.2.2 运行所有测试并确认最后一个未通过

确认最后一个测试未通过后，就能断定它不会在没有引入新代码的情况下错误通过。如果这

个测试通过了，就意味着要么相关功能早就存在，要么测试本身存在误报问题。如果测试无论怎样实现都能通过，就意味着它毫无价值，应该删除。

最后一个测试不仅必须未通过，还必须是预期原因导致的。

在这个阶段，我们依然处于红灯状态：运行测试，但最后一个未通过。

3.2.3 编写实现代码

这个阶段的目标是编写代码使最后一个测试通过。不要试图让代码完美无缺，也不要为编写花过多时间。即便编写的不好或者不是最后的，也没有关系，后面还有改进的机会。我们的真实意图是打造一个由测试构成的安全网，并确认这些测试都能通过。不要试图引入最后一个测试未描述的功能。要想引入新功能，必须回到第一步，先编写新测试。然而，仅当所有既有测试都通过后，我们才能这么做。

在这个阶段，我们依然处于红灯状态。虽然已编写的代码可能让所有测试都通过，但这种假设还未得到证实。

3.2.4 运行所有测试

应运行所有测试，而不是只运行最后编写的那个测试，这至关重要。你刚编写的代码可能让最后一个测试得以通过，但同时破坏了其他功能。通过运行所有测试，不仅可确认最后一个测试的实现是正确的，还可确认它没有破坏整个应用程序的完整性。如果整个测试集执行速度缓慢，就昭示着测试编写得不好或者代码耦合度太高。耦合度太高将导致难以隔离外部依赖，进而增加执行测试所需的时间。

在这个阶段，我们处于绿灯状态：所有测试都通过，且应用程序的行为符合预期。

3.2.5 重构

前面所有步骤都是必不可少的，但这一步是可选的。虽然很少在每个周期结束后都进行重构，但迟早需要甚至必须这样做。并非每个测试的实现都需要重构；没有明确的规定说什么时候该重构、什么时候不用重构。一旦认为可以更佳或更优的方式重写代码，那就是重构的最佳时机。

什么样的代码需要重构呢？这个问题不好回答，因为重构的原因有很多：代码难以理解、代码位置不合理、代码重复、名称没有清晰阐述意图、方法太长、类的功能太多等——这个清单可不断列下去。不管原因是什么，最重要的规则是重构不能改变任何既有功能。

3.2.6　重复

所有步骤都完成后（其中重构是可选的），再重复它们。乍一看，整个过程好像太长、太复杂，但并不是这样的。经验丰富的TDD践行者编写1~10行代码后就切换到下一步，因此整个周期的持续时间为几秒~几分钟。如果更长，就说明测试的范围太大，应将其分成多个更小的测试。一定要快速前进，快速失败并更正，然后再重复。

深入了解“红灯–绿灯–重构”过程后，下面确定我们要使用这种过程开发的应用程序的需求。

3.3　“井字游戏”的需求

“井字游戏”是儿童常玩的一种游戏，规则非常简单。

> “井字游戏”是两个人使用纸和铅笔玩的一种游戏，双方轮流在一个3×3的网格中画X和O，最先在水平、垂直或对角线上将自己的3个标记连起来的玩家获胜。
>
> 有关这款游戏的更详细信息，请参阅维基百科（http://en.wikipedia.org/wiki/Tic-tac-toe）。

更详细的需求将在后面介绍。

这个练习中，你将根据需求编写测试，再编写满足测试期望的代码。最后，如果必要，将对代码进行重构。你将重复这个过程，针对同一需求编写多个测试。对针对当前需求编写的测试和实现代码满意后，进入下一个需求，直到处理完所有需求。

现实世界中，你并不会预先制定如此详细的需求，而是直接编写同时用作需求和验证的测试。然而，熟练掌握TDD前，我们必须将需求和测试分开定义。

虽然后面提供了所有测试和实现，但每次只阅读一个需求，再自己尝试编写测试和实现代码。完成后再将解决方案与本书提供的解决方案进行比较，然后进入下一个需求。不存在有且只有一个解决方案的情况——你的解决方案可能比这里提供的更好。

3.4　开发“井字游戏”

准备好开始编写代码了吗？下面看第一个需求。

3.4.1　需求 1

我们应该首先定义边界，以及将棋子放在哪些地方非法。

> 可将棋子放在3×3棋盘上任何没有棋子的地方。

可将这个需求分成三个测试：

- 如果棋子放在超出了X轴边界的地方，就引发RuntimeException异常；
- 如果棋子放在超出了Y轴边界的地方，就引发RuntimeException异常；
- 如果棋子放在已经有棋子的地方，就引发RuntimeException异常。

正如你看到的，与第一个需求相关的测试都验证输入参数。至于这些棋子该如何处理呢？这个需求什么都没说。

编写第一个测试前，先简单说说如何使用JUnit测试异常。

JUnit 4.7引入了一项名为**规则**（**Rule**）的功能，使用它可以做很多不同的事情（更详细的信息请参阅https://github.com/junit-team/junit/wiki/Rules），但在这里我们感兴趣的是规则ExpectedException：

```
public class FooTest {
  @Rule
  public ExpectedException exception =
    ExpectedException.none();

  @Test
  public void whenDoFooThenThrowRuntimeException() {
    Foo foo = new Foo();
    exception.expect(RuntimeException.class);
    foo.doFoo();
  }
}
```

这个示例中，我们指定ExpectedException是一条规则；接下来，在测试doFooThrowsRuntimeException中，我们指出Foo类被实例化后，期望引发RuntimeException异常。如果这种异常是在实例化前引发的，这个测试将失败；反之，测试将成功。

@Before可用来标注要在每个测试前运行的方法，这是一项很有用的功能。例如，你可使用它实例化测试中使用的类，或者指定要在每个测试前执行的其他操作：

```
private Foo foo;

@Before
public final void before() {
```

```
        foo = new Foo();
    }
```

这个示例中，在每个测试前都将实例化Foo类，这样就不用在每个测试方法中重复编写实例化Foo的代码了。

每个测试方法都必须用@Test标注，让JunitRunner知道哪些方法是测试。测试以随机顺序运行，因此务必确保每个测试都是自给自足的，不依赖于其他测试设置的状态：

```
@Test
public void whenSomethingThenResultIsSomethingElse() {
  // 这是一个测试方法。
}
```


有了这些知识后，你应该能够编写第一个测试，并接着编写其实现。完成后将其与后面提供的解决方案进行比较。

> **给测试方法指定描述性名称**
>
> 这样做的好处之一是有助于理解测试的目标。
>
> 使用描述测试的方法名很有益，可帮助掌握有些测试失败的原因以及在什么情况下增加测试可提高代码覆盖率。在测试方法名中，应明确指出测试前设置的条件、执行的操作以及期望的结果。
>
> 给测试方法命名的方式很多，我喜欢采用BDD场景使用的given/when/then语法给它们命名，其中Given描述前置条件，When描述操作，而Then描述期望的结果。如果测试没有前置条件（这些条件通常是使用注解@Before和@BeforeClass设置的），则可省略Given。
>
> 不要完全依靠注释指出测试的目标，因为使用IDE执行测试时，注释不会出现，它们也不会出现在CI工具或构建工具生成的报告中。

除编写测试外，你还需要运行它们。由于我们使用的是Gradle，因此要运行测试，可从命令提示符执行如下命令：

```
$ gradle test
```

IntelliJ IDEA提供了一个极佳的Gradle任务模型，可通过选择菜单**View > Tool Windows > Gradle**访问。它列出了使用Gradle可运行的所有任务，其中一个是测试。

如何运行测试由你决定——可使用你认为合适的任何方式，只要确保运行所有测试即可。

1. 测试

首先检查棋子是否放在3×3棋盘的边界内：

```
package com.packtpublishing.tddjava.ch03tictactoe;
```

```
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class TicTacToeSpec {

    @Rule
    public ExpectedException exception =
    ExpectedException.none();
    private TicTacToe ticTacToe;

    @Before
    public final void before() {
        ticTacToe = new TicTacToe();
    }

    @Test
    public void whenXOutsideBoardThenRuntimeException()
        {
     exception.expect(RuntimeException.class);
     ticTacToe.play(5, 2);

    }

}
```

> 棋子放在超出X轴边界的地方时，将引发`RuntimeException`异常。

这个测试中，我们指出调用方法`ticTacToe.play(5, 2)`时，期望的结果是引发`RuntimeException`异常。这个测试既简短又容易，要让它通过应该也很容易：只需创建方法`play`，并确保它在参数`x`小于1或大于3（棋盘是3×3的）时引发`RuntimeException`异常。你应运行这个测试三次：第一次运行时，它应该不能通过，因为此时还没有方法`play`；添加这个方法后，测试也应不能通过，因为它没有引发异常`RuntimeException`；第三次运行时应该通过，因为实现了与这个测试相关联的所有代码。

2. 实现

明确什么情况下应引发异常后，实现代码编写起来应该很简单：

```
package com.packtpublishing.tddjava.ch03tictactoe;

public class TicTacToe {

    public void play(int x, int y) {
        if (x < 1 || x > 3) {
            throw
```

```
            new RuntimeException("X is outside board");
        }
    }

}
```

正如你看到的，这里只包含让测试能够通过的最少代码，而没有任何多余代码。

> 有些TDD践行者从字面意思上解读“最少”，让方法play只包含代码行throw newRuntimeException();。我通常将“最少”理解为“在合理范围内尽可能少”。

这里没有将数字相加，也没有返回任何值，其根本目标是以极快的速度做细微的修改（还记得前面提到的乒乓球运动吗？）。至此，我们完成了“红灯–绿灯”部分，但这些代码的改进空间不大，因此不重构。

下面进入下一个测试。

3. 测试

这个测试与前一个测试几乎相同，但验证的是Y轴：

```
@Test
public void whenYOutsideBoardThenRuntimeException() {
    exception.expect(RuntimeException.class);
    ticTacToe.play(2, 5);
}
```

> 棋子放在超出Y轴边界的地方时，将引发RuntimeException异常。

4. 实现

这个规范的实现几乎与前一个相同，只需在参数Y不在指定范围内时引发异常即可：

```
    public void play(int x, int y) {
    if (x < 1 || x > 3) {
        throw
        new RuntimeException("X is outside board");
    } else if (y < 1 || y > 3) {
        throw
        new RuntimeException("X is outside board");
    }
}
```

为让最后一个测试通过，添加一条“检查参数Y是否在棋盘内”的else子句。

下面编写当前需求涉及的最后一个测试。

5. 测试

确定棋子在棋盘边界内后，还需确保它放在未被别的棋子占据的地方：

```
@Test
public void whenOccupiedThenRuntimeException() {
    ticTacToe.play(2, 1);
    exception.expect(RuntimeException.class);
    ticTacToe.play(2, 1);
}
```

> 棋子放在被别的棋子占据的地方时，将引发RuntimeException异常。

这就是最后一个测试。编写实现后，即可认为第一个需求完成了。

6. 实现

为实现最后一个测试，应将既有棋子的位置存储在一个数组中。每当玩家放置新棋子时，都应确认棋子放在未占用的位置，否则引发异常：

```
private Character[][] board = {{'\0', '\0', '\0'},
{'\0', '\0', \0'}, {'\0', '\0', '\0'}};

public void play(int x, int y) {
    if (x < 1 || x > 3) {
        throw
        new RuntimeException("X is outside board");
    } else if (y < 1 || y > 3) {
        throw
        new RuntimeException("Y is outside board");
    }
    if (board[x - 1][y - 1] != '\0') {
        throw
        new RuntimeException("Box is occupied");
    } else {
        board[x - 1][y - 1] = 'X';
    }
}
```

我们检查要放置棋子的位置是否被占用，如果未占用，就将相应数组元素的值从空（\0）改为被占用（X）。注意，我们还没有记录棋子是谁（X还是O）的。

7. 重构

这些代码虽然满足了测试指定的需求，但有点令人迷惑。如果有人阅读这些代码，会搞不清楚方法play的目的。应重构这个方法，将其中的代码放在多个方法中。重构后的代码类似于下面这样：

```
public void play(int x, int y) {
    checkAxis(x);
    checkAxis(y);
    setBox(x, y);
}

private void checkAxis(int axis) {
    if (axis < 1 || axis > 3) {
        throw
        new RuntimeException("X is outside board");
    }
}

private void setBox(int x, int y) {
    if (board[x - 1][y - 1] != '\0') {
        throw
        new RuntimeException("Box is occupied");
    } else {
        board[x - 1][y - 1] = 'X';
    }
}
```


这个重构过程中，没有改变方法`play`的功能，其行为与以前完全相同，但代码的可读性更强了。由于我们有覆盖了所有功能的测试，因此不用害怕重构时犯错。只要确保所有测试都通过且重构时没有引入新行为，就可以放心大胆地修改代码。

完整的源代码可在Git仓库`tdd-java-ch03-tic-tac-toe`的分支`01-exceptions`（https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/01-exceptions）中找到。

3.4.2 需求 2

现在处理轮到哪个玩家落子的问题。

> 需要提供一种途径，用于判断接下来该谁落子。

可将这个需求分成三个测试：

- 玩家`X`先下；
- 如果上一次是`X`下的，接下来将轮到`O`下；
- 如果上一次是`O`下的，接下来将轮到`X`下。

到目前为止，我们还未使用过JUnit断言。要使用断言，需要导入`org.junit.Assert`类中的静态（`static`）方法：

```
import static org.junit.Assert.*;
```

Assert类中的方法都非常简单，它们大都以assert打头。例如，assertEquals对两个对象进行比较：assertNotEquals验证两个对象不同，而assertArrayEquals验证两个数组相同。这两个断言都有很多重载版本，因此几乎能够对任何类型的Java对象进行比较。

在这里，我们需要比较两个字符，其中第一个是预期的字符，而第二个是方法nextPlayer返回的实际字符。

现在编写这些测试及其实现。

> **先编写测试，再编写实现代码**
>
> 这样做的好处是：可确保编写的代码是可测试的，且每行代码都有对应的测试。
>
> 通过先编写或修改测试，开发人员可在编写代码前专注于需求。这是与完成实现后再编写测试的主要差别所在。测试先行的另一个好处是，可避免原本应为质量保证的测试沦为质量检查。

1. 测试

玩家X先下：

```
@Test
public void givenFirstTurnWhenNextPlayerThenX() {
    assertEquals('X', ticTacToe.nextPlayer());
}
```

> 应该是玩家X先下。

这个测试应该是不言自明的：我们期望nextPlayer返回X。如果现在运行这个测试，将发现它都不能通过编译，这是因为还没有方法nextPlayer。我们的任务是编写方法nextPlayer，并确保它返回正确的值。

2. 实现

其实根本不需要检查玩家X是否先下，因为就目前而言，只需让nextPlayer返回X就能让这个测试通过。后面的测试将要求我们修改这个方法的代码：

```
public char nextPlayer() {
    return 'X';
}
```

3. 测试

现在需要确保让玩家轮流下。玩家X下棋后，应轮到玩家O，然后再轮到玩家X，以此类推：

```
@Test
public void givenLastTurnWasXWhenNextPlayerThenO()
{
    ticTacToe.play(1, 1);
    assertEquals('O', ticTacToe.nextPlayer());
}
```

> 如果前一次是玩家X下的，接下来应轮到玩家O。

4. 实现

为跟踪接下来该谁下，需要存储前一次下棋的玩家：

```
private char lastPlayer = '\0';

public void play(int x, int y) {
    checkAxis(x);
    checkAxis(y);
    setBox(x, y);
    lastPlayer = nextPlayer();
}

public char nextPlayer() {
    if (lastPlayer == 'X') {
        return 'O';
    }
    return 'X';
}
```

你很可能已经进入状态。测试很小且易于编写，有了足够的经验后，编写一个测试只需一分钟甚至几秒钟；而编写实现所需的时间也差不多，甚至更短。

5. 测试

我们终于可以检查玩家O下后是不是轮到玩家X了。

> 如果前一次是玩家O下的，接下来应是玩家X下。

即使什么都不用做，这个测试也能通过。因此它毫无用处，应当删除。如果编写这个测试，将发现它存在错报问题：在没有修改实现的情况下就能通过。你可以自己试一试。编写测试后，如果它在没有编写任何实现代码时就能通过，应将其删除。

可在Git仓库`tdd-java-ch03-tic-tac-toe`的分支`02-next-player`（https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/02-next-player）中找到源代码。

3.4.3 需求 3

现在考虑这个游戏的获胜规则。相比于前面的代码，这部分工作更繁琐。我们必须检查所有可能获胜的情况，只要满足其中一个，就宣布相应玩家获胜。

> 最先在水平、垂直或对角线上将自己的3个标记连起来的玩家获胜。

要检查同一玩家的3颗棋子是否连成了线，需要检查水平方向、垂直方向和对角线。

1. 测试

下面首先定义方法play的默认返回值：

```
@Test
public void whenPlayThenNoWinner()
{
  String actual = ticTacToe.play(1,1);
  assertEquals("No winner", actual);
}
```

> 如果不满足获胜条件，则无人获胜。

2. 实现

默认返回值总是最容易实现的，这里也不例外：

```
public String play(int x, int y) {
    checkAxis(x);
    checkAxis(y);
    setBox(x, y);
    lastPlayer = nextPlayer();
    return "No winner";
}
```

3. 测试

指定默认结果（没有人获胜）后，处理各种获胜条件：

```
@Test
public void whenPlayAndWholeHorizontalLineThenWinner() {
    ticTacToe.play(1, 1); // X
    ticTacToe.play(1, 2); // O
    ticTacToe.play(2, 1); // X
    ticTacToe.play(2, 2); // O
    String actual = ticTacToe.play(3, 1); // X
    assertEquals("X is the winner", actual);
}
```

> 一个玩家的棋子占据整条水平线就赢了。

4. 实现

为让这个测试通过，需要检查是否有水平线全被当前玩家的棋子占据。到目前为止，我们根本不关心存储到数组board中的值是什么，但现在不但要记录哪些棋盘格是空的，还需记录各个棋盘格被哪个玩家占据：


```
    public String play(int x, int y) {
        checkAxis(x);
        checkAxis(y);
        lastPlayer = nextPlayer();
        setBox(x, y, lastPlayer);
        for (int index = 0; index < 3; index++) {
            if (board[0][index] == lastPlayer &&
                    board[1][index] == lastPlayer &&
                    board[2][index] == lastPlayer) {
                return lastPlayer + " is the winner";
            }
        }
        return "No winner";
    }
    private void setBox(int x, int y, char lastPlayer)
{
    if (board[x - 1][y - 1] != '\0') {
         throw
         new RuntimeException("Box is occupied");
    } else {
         board[x - 1][y - 1] = lastPlayer;
    }
}
```

5. 重构

前面的代码能够让测试通过，完成了尽快让测试通过的使命，但并非没有改进的空间。现在我们有了确保预期行为完整性的测试，可对代码进行重构：

```
    private static final int SIZE = 3;
    public String play(int x, int y) {
        checkAxis(x);
        checkAxis(y);
        lastPlayer = nextPlayer();
        setBox(x, y, lastPlayer);
        if (isWin()) {
            return lastPlayer + " is the winner";
        }
        return "No winner";
    }
```

```
private boolean isWin() {
for (int i = 0; i < SIZE; i++) {
     if (board[0][i] + board[1][i] + board[2][i]
      == (lastPlayer * SIZE)) {
         return true;
        }
    }
    return false;
}
```

重构后的解决方案看起来更好。play依然很短，很容易理解。将实现获胜逻辑的代码移到一个独立的方法中，不仅让方法play的目的变得清晰，还能让我们独立添加检查获胜条件的代码。

6. 测试

我们还需检查是否有垂直线完全被某个玩家占据：

```
@Test
public void whenPlayAndWholeVerticalLineThenWinner() {
    ticTacToe.play(2, 1); // X
    ticTacToe.play(1, 1); // O
    ticTacToe.play(3, 1); // X
    ticTacToe.play(1, 2); // O
    ticTacToe.play(2, 2); // X
    String actual = ticTacToe.play(1, 3); // O
    assertEquals("O is the winner", actual);
}
```

> 一个玩家的棋子占据整条垂直线就赢了。

7. 实现

这个实现应该与前一个类似。前面在水平方向上做了检查，现在需要在垂直方向上做同样的检查：

```
private boolean isWin() {
 int playerTotal = lastPlayer * 3;
 for (int i = 0; i < SIZE; i++) {
     if (board[0][i] + board[1][i] + board[2][i]
     == playerTotal) {
         return true;
     } else if (playerTotal == )
  {
         return true;
     }
  }
 return false;
}
```

8. 测试

水平线和垂直线都处理后，该将注意力转向对角线了：

```
@Test
public void whenPlayAndTopBottomDiagonalLineThenWinner() {
    ticTacToe.play(1, 1); // X
    ticTacToe.play(1, 2); // O
    ticTacToe.play(2, 2); // X
    ticTacToe.play(1, 3); // O
    String actual = ticTacToe.play(3, 3); // O
    assertEquals("X is the winner", actual);
}
```


> 一个玩家的棋子占据从左上角到右下角的整条对角线就赢了。

9. 实现

由于这里只涉及一条线，因此可直接检查，无需使用循环：


```
private boolean isWin() {
 int playerTotal = lastPlayer * 3;
 for (int i = 0; i < SIZE; i++) {
     if (board[0][i] + board[1][i] + board[2][i]
     playerTotal) {
         return true;
     } else if (playerTotal == )
{
         return true;
     }
}
  if ((board[0][0] + board[1][1] + board[2][2])
  == playerTotal) {
     return true;
 }
  return false;
}
```

10. 测试

最后，还有最后一个可能的获胜条件需要处理：

```
@Test
public void whenPlayAndBottomTopDiagonalLineThenWinner() {
    ticTacToe.play(1, 3); // X
    ticTacToe.play(1, 1); // O
    ticTacToe.play(2, 2); // X
    ticTacToe.play(1, 2); // O
    String actual = ticTacToe.play(3, 1); // O
    assertEquals("X is the winner", actual);
}
```

一个玩家的棋子占据从左下角到右上角的整条对角线就赢了。

11. 实现

这个测试的实现应该与前一个几乎完全相同：

```
private boolean isWin() {
 int playerTotal = lastPlayer * 3;
 for (int i = 0; i < SIZE; i++) {
     if (board[0][i] + board[1][i] + board[2][i]
     playerTotal) {
         return true;
     } else if (playerTotal == )
{
         return true;
     }
 }
 if ((board[0][0] + board[1][1] + board[2][2])
 == playerTotal) {
     return true;
 } else if (playerTotal == (board[0][2] + board[1][1] +
   board[2][0])) {
     return true;
  }
  return false;
}
```

12. 重构

处理对角线时，所做的计算看起来不太好，也许重用既有的循环更合适：

```
private boolean isWin() {
    int playerTotal = lastPlayer * 3;
    char diagonal1 = '\0';
    char diagonal2 = '\0';
    for (int i = 0; i < SIZE; i++) {
        diagonal1 += board[i][i];
        diagonal2 += board[i][SIZE - i - 1];
        if (board[0][i] + board[1][i] + board[2][i]) ==
          playerTotal) {
            return true;
        } else if (playerTotal == ) {
            return true;
        }
    }
    if (diagonal1 == playerTotal || diagonal2 == playerTotal) {
        return true;
    }
    return false;
}
```

可在Git仓库`tdd-java-ch03-tic-tac-toe`的分支`03-wins`（https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/03-wins）中找到源代码。

下面处理最后一个需求。

3.4.4　需求 4

现在缺失的唯一一项内容是如何处理平局。

> 所有格子都占满则为平局。

1. 测试

可以通过填满棋盘的所有格子测试平局结果：

```
@Test
public void whenAllBoxesAreFilledThenDraw() {
    ticTacToe.play(1, 1);
    ticTacToe.play(1, 2);
    ticTacToe.play(1, 3);
    ticTacToe.play(2, 1);
    ticTacToe.play(2, 3);
    ticTacToe.play(2, 2);
    ticTacToe.play(3, 1);
    ticTacToe.play(3, 3);
    String actual = ticTacToe.play(3, 2);
    assertEquals("The result is draw", actual);
}
```

2. 实现

检查是否为平局非常简单——只需检查是否已占满整个棋盘。为此，可遍历数组board：

```
public String play(int x, int y) {
    checkAxis(x);
    checkAxis(y);
    lastPlayer = nextPlayer();
    setBox(x, y, lastPlayer);
    if (isWin()) {
        return lastPlayer + " is the winner";
    } else if (isDraw()) {
        return "The result is draw";
    } else {
        return "No winner";
    }
}

private boolean isDraw() {
```

```
        for (int x = 0; x < SIZE; x++) {
            for (int y = 0; y < SIZE; y++) {
                if (board[x][y] == '\0') {
                    return false;
                }
            }
        }
        return true;
    }
```

3. 重构

虽然方法`isWin`与最后一个测试无关，但也可重构。例如，我们无需检查所有获胜条件，而只需检查与最后一个棋子的位置相关的获胜条件。最终的版本类似于下面这样：

```
    private boolean isWin(int x, int y) {
        int playerTotal = lastPlayer * 3;
        char horizontal, vertical, diagonal1, diagonal2;
        horizontal = vertical = diagonal1 = diagonal2 = '\0';
        for (int i = 0; i < SIZE; i++) {
            horizontal += board[i][y - 1];
            vertical += board[x - 1][i];
            diagonal1 += board[i][i];
            diagonal2 += board[i][SIZE - i - 1];
        }
        if (horizontal == playerTotal
                || vertical == playerTotal
                || diagonal1 == playerTotal
                || diagonal2 == playerTotal) {
            return true;
        }
        return false;
    }
```

可随时对代码的任何一部分进行重构，只要此时所有测试均通过。通常代码编写后立即进行重构最容易也最快，但重构几天、几月甚至几年前编写的代码更可贵。发现可让代码更好就是重构它的最佳时机，至于代码是谁写的以及什么时候写的都不重要，毕竟让代码变得更好总是一件值得去做的好事。

可在Git仓库`tdd-java-ch03-tic-tac-toe`的分支`04-draw`（https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/04-draw）中找到源代码。

3.5　代码覆盖率

前面的练习中，我们没有使用代码覆盖率工具，原因是我们想让你专注于“红灯–绿灯–重构”过程。编写一个测试，发现它不能通过；编写实现代码，发现所有测试都通过；只要有机会就重构代码使其变得更好，并重复这个过程。前面的测试覆盖了所有的情形吗？诸如JaCoCo等**代码覆**

盖率工具可回答这个问题。应该使用这些工具吗？也许仅在刚开始使用TDD时才使用。下面说明其中原因。刚开始使用TDD时，你很可能遗漏一些测试，或实现的代码比测试要求的多。这些情况下，使用代码覆盖率工具是一种从自己的错误中学习的极佳方式。随着TDD使用经验日益丰富，你会越来越不需要这样的工具。届时编写测试以及刚好能让它们通过的代码时，无论有没有JaCoCo这样的工具，代码覆盖率都会很高。肯定有一小部分代码未被测试覆盖，因为你能够针对哪些代码不值得测试做出明智的决定。

JaCoCo等工具主要设计用于验证实现代码后编写的测试是否提供了足够的覆盖率。TDD中，我们采取反转的做法，即先编写测试，再编写实现。

然而，我们依然建议你将JaCoCo作为学习工具。至于以后是否使用，由你自己决定。

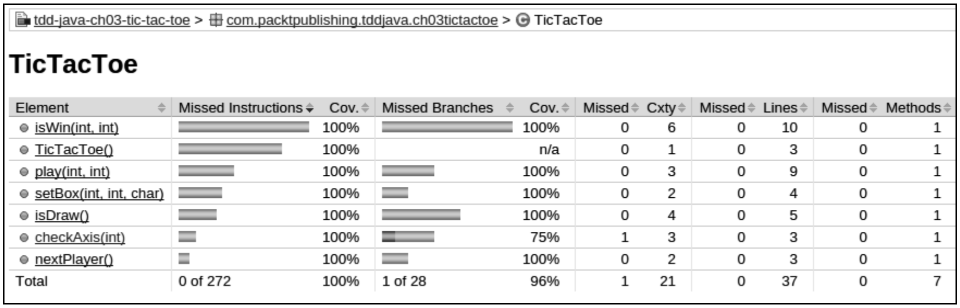
要想在Gradle中启用JaCoCo，可在文件`build.gradle`中添加如下内容：

```
apply plugin: 'jacoco'
```

这样，每当运行测试时，Gradle都将收集JaCoCo指标。使用Gradle目标（target）`jacocoTestReport`可将这些指标转换为漂亮的报告。下面再次运行前面的测试，看看代码覆盖率有多高：

```
$ gradle clean test jacocoTestReport
```

最终结果存储在目录`build/reports/jacoco/test/html`下的报告中。结果根据这个练习中实现的解决方案而异；我得到的结果表明，指令覆盖率为100%，而分支覆盖率为96%。还有4%的分支未覆盖，这是因为没有测试玩家将棋子坐标指定为0或负数的情形。实现代码考虑了这样的情形，但没有覆盖它的测试。总体而言，覆盖率还是非常高的。



tdd-java-ch03-tic-tac-toe > com.packtpublishing.tddjava.ch03tictactoe > TicTacToe

TicTacToe

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isWin(int, int)		100%		100%	0	6	0	10	0	1
TicTacToe()		100%		n/a	0	1	0	3	0	1
play(int, int)		100%		100%	0	3	0	9	0	1
setBox(int, int, char)		100%		100%	0	2	0	4	0	1
isDraw()		100%		100%	0	4	0	5	0	1
checkAxis(int)		100%		75%	1	3	0	3	0	1
nextPlayer()		100%		100%	0	2	0	3	0	1
Total	0 of 272	100%	1 of 28	96%	1	21	0	37	0	7

JaCoCo将被加入源代码，这些源代码可在Git仓库`tdd-java-ch03-tic-tac-toe`的分支`05-jacoco`（https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/05-jacoco）中找到。

3.6 更多练习

前面开发了“井字游戏”的最常用版本，作为额外练习，请从维基百科（http://en.wikipedia.

org/wiki/Tic-tac-toe）选择这个游戏的其他版本，并使用“红灯–绿灯–重构”流程实现。完成后，通过实现AI让计算机充当玩家O。由于“井字游戏”通常以平局告终，因此你只要实现这样的AI即可，即不管玩家X怎么走都至少能打成平局。

完成这些练习时，一定要快速前进，就像打乒乓球一样。另外，最重要的是，别忘了使用“红灯–绿灯–重构”流程。

3.7　小结

我们使用“红灯–绿灯–重构”流程完成了“井字游戏”，这些示例本身都很简单，易于理解。

本章并非要深入探讨复杂的东西（后面会介绍这些内容），而是要让你养成反复使用“红灯–绿灯–重构”流程的习惯。

你学习了如下内容：开发软件的最简单方式是将其分成小块；设计方案脱胎于测试，而不是预先采用复杂的方法进行制定；先编写测试并确定未通过后，再着手编写实现代码；确定最后一个测试未通过后，就能肯定它是有效的（你一不小心就会犯错，编写总是能够通过的测试），要实现的功能还不存在；测试未通过后，编写其实现代码；编写实现时，力图使其尽可能简单，只要能让测试通过就行，而不试图提供完美的解决方案；不断重复这个过程，直到认为需要对代码进行重构为止；重构时不能引入任何新功能（即不改变应用程序的行为），而只是对代码进行改进，使其更容易理解和维护。

下一章将详细介绍TDD中的单元，以及如何根据这些单元创建测试。

第 4 章

单元测试——专注于当下而非过往

“要打造出出类拔萃的作品，你必须专注于最细小的细节。”

——乔治·阿玛尼

前面说过，每章都将探索一个不同的Java测试框架。这一章也不例外，我们将使用TestNG制定规范。

在前一章，我们练习了“红灯–绿灯–重构”过程。虽然使用了单元测试，但未深入阐述单元测试在TDD中的工作原理。本章将拓展前一章介绍的知识，深入阐述单元测试的定义及其在软件开发方法TDD中扮演的角色。

本章旨在让你学会如何专注于当前要开发的单元，并忽略或隔离已完成的单元。

熟悉TestNG和单元测试后，我们将深入介绍下一个应用程序的需求并开始编写代码。

本章涵盖如下主题：

- 单元测试；
- TDD中的单元测试；
- TestNG；
- “遥控军舰”的需求；
- 开发“遥控军舰”；
- 小结。

4.1 单元测试

除非系统非常小，否则很难频繁地进行手工测试。要避免此类操作，唯一的办法是使用自动化测试；要缩短和降低构建、部署和维护应用程序的时间和成本，唯一有效的方法是使用自动化

测试。为卓有成效地管理应用程序，实现和测试代码必须尽可能简单，这至关重要。简约（http://www.extremeprogramming.org/rules/simple.html）是极限编程（XP）的核心价值观之一，也是TDD和一般性编程的关键所在；这通常是通过将系统分成细小的单元实现的。在Java中，单元就是方法。作为最小的编程单位，单元提供的反馈环路是最快的，因此我们的大部分时间都花在思考和处理它们上。与实现方法相对应的是单元测试，它们在测试中的分量最重。

4.1.1　何为单元测试

单元测试（UT）是一种实践，要求我们对每个隔离的小型代码单元进行测试。单元通常是方法，但有些情况下，整个类乃至整个应用程序都可视为单元。要编写UT，需要将受测代码同应用程序的其他部分隔离。最理想的情况是，要么系统已实现这样的隔离，要么可通过使用模拟对象实现隔离（模拟对象将在第6章更深入地介绍）。如果特定方法的单元测试跨越了该单元的边界，它将变成集成测试。这种情况下，测试的是哪些代码将变得不那么清晰。如果测试失败，问题的范围将急剧增大，找出原因的工作将更为繁琐。

4.1.2　为何要进行单元测试

一个常见的问题是：为何使用单元测试而不是功能和集成测试。在严重依赖手动测试的组织中，这个问题尤为突出。然而，这个问题本身就有问题。单元测试并非要取代其他类型的测试，而只是缩小其他测试的范围。从本质上说，单元测试的编写比其他任何类型的测试都更容易、更快捷，从而能够降低成本、缩短上市时间。由于编写并运行单元测试所需的时间更少，它们通常能够更快发现问题。而问题发现得越早，修复的成本就越低。Bug出现后，如果能够在几分钟内发现，则与几天、几周乃至几个月后才发现相比，修复将容易得多。

4.1.3　代码重构

代码重构指的是对既有代码的结构进行修改，同时不改变其外部行为。重构旨在改进既有代码，这样做的原因很多：提高可读性、降低复杂度、使其更易于维护或更容易扩展等。不管重构的原因是什么，其终极目标都是改进代码的某个方面，从而降低技术债务：减少因设计、架构或编码不佳而需要做的额外工作。

通常，我们在不改变行为的情况下做一系列细微的修改以实现重构。通过缩小修改范围，我们可始终确认所做的修改没有破坏既有功能。而要获得这样的确认，唯一有效的办法是使用自动化测试。

单元测试的一大优点是，为重构提供最有力的支持。如果没有自动化测试确认应用程序依然像期望的那样工作，重构将风险重重。虽然任何类型的测试都可用于提供重构所需的代码覆盖率，

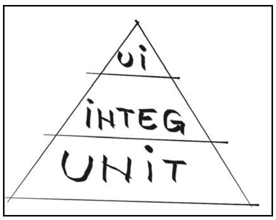
但大多数情况下，只有单元测试能够达到要求的细致程度。

4.1.4　为何不只使用单元测试

此时你可能会问，单元测试能够满足所需的测试需求吗？不幸的是，答案是否定的。虽然单元测试满足的测试需求通常是最多的，但功能测试和集成测试也不可或缺。

其他类型的测试将在本书后面更详细地介绍，这里先来说说单元测试和其他测试的几个重要差别：

- **单元测试**旨在对小型功能单元进行检查。在Java中，这些单元就是方法。对于所有外部依赖，诸如对其他类、方法或数据库的调用等，都应在内存中完成，这是通过使用模拟对象、存根、间谍、伪造对象和哑元对象实现的。杰拉德·梅萨罗斯发明了一个更通用的术语——测试替身（http://en.wikipedia.org/wiki/Test_double），它涵盖了前述各种对象。单元测试很简单、易于编写且运行速度很快，通常在所有测试中占据的分量最大。
- **功能测试和验收测试**的职责是核实整个应用程序像预期的那样工作。这两种测试的用途不同，但目标相似。单元测试旨在检查代码的内部质量，而功能测试和验收测试用于确保整个系统在客户或用户看来能够正常工作。为编写和运行这些测试，需要付出更多成本和劳动，因此其数量通常比单元测试少。
- **集成测试**旨在核实各个单元、模块、应用程序乃至系统被妥善地集成在一起。你可能有一个前端应用程序，它使用后端API，而后端API又与一个数据库通信。这种情况下，集成测试的职责是核实这三个不同的组件被紧密地集成在一起，能够彼此通信。执行集成测试前，已确认所有单元都能正常工作、所有功能测试和验收测试都已通过，因此集成测试唯一的职责是确认所有组件能够很好地协同工作，所以其数量是最少的。



这个测试金字塔表明，单元测试的数量比高层测试（UI测试、集成测试等）多得多。为何会这样呢？单元测试的编写更容易，运行更快，提供的代码覆盖率更高。比如注册功能，我们应测试下述情形下的结果：用户名为空、密码为空、用户名或密码的格式不正确、用户名已被占用等。仅为测试这一项功能，就可能需要数十乃至数百个测试；如果通过UI运行所有这些测试，代价将非常高（需要花很多时间编写且运行缓慢）。相反，执行这种验证的单元测试很容易编写且运行速度快。如果使用单元测试覆盖所有这些情形，我们就只需编写一个集成测试——检查UI是否调用了正确的后端方法。从集成的角度看，细节已无关紧要，因为我们知道单元级已覆盖所有情形。

4.2　TDD 中的单元测试

TDD中单元测试的编写方式有何不同呢？主要是编写时机。传统做法是在实现代码完成后编写单元测试，而TDD中的顺序相反——先编写测试。未使用TDD的情况下，单元测试用于验证既有代码；而在TDD中，应将单元测试作为驱动开发和设计的动力，它们定义最小可能单元的行为，指定有待实现的微型需求。测试指出了你接下来该做什么以及该做到什么程度为止，至于要完成的工作量，则随测试类型（单元测试、功能测试、集成测试等）而异。TDD中，单元测试指定接下来应完成尽可能小的任务，即一个方法乃至其一部分。另外，TDD还要求我们遵守一些设计原则，如**KISS**（**keep it simple stupid**，**保持简单**）。通过编写范围很小的简单测试，可确保这些测试的实现也同样简单。通过要求测试不使用外部依赖，可确保实现代码严格遵守关注点分离原则。有关TDD如何帮助我们编写更好的代码，还有很多其他的例子，而仅使用单元测试是无法带来这些好处的。未使用TDD的情况下，单元测试将只用于测试既有代码，对设计毫无影响。

总之，未使用TDD的情况下，单元测试的主要目标是验证既有代码；而在TDD中，单元测试是预先编写的，其主要目标是定义需求和设计，而验证只是副产品。与实现后再编写测试相比，这样做的一个结果是产品质量更高。

TDD迫使我们详细地考虑需求和设计、编写整洁而可行的代码，以及创建可执行的需求并频繁重构。另外，这样编写的单元测试的代码覆盖率极高，每当对代码进行修改后，都可使用它们进行回归测试。未使用TDD的情况下，单元测试只是测试，其质量也是不确定的。

4.3　TestNG

JUnit和TestNG是两个主要的Java测试框架。前一章已使用JUnit编写过测试，大家很可能对其工作原理有深入了解。TestNG又如何呢？它是为改进JUnit而开发的，同时提供了一些JUnit没有的功能。

接下来的几小节将总结这两个框架的一些差别。阐述这些差别的同时，我们还将力图从TDD单元测试角度对这两个框架进行评价。

4.3.1　注解`@Test`

JUnit和TestNG都使用注解`@Test`将方法指定为测试。JUnit要求使用`@Test`对每个用作测试的方法进行注解，而TestNG同时允许在类级使用这个注解。以这种方式使用该注解时，除非特别指定，否则类中所有公有方法都被视为测试：

```
@Test
public class DirectionSpec {
```

```
    public void whenGetFromShortNameNThenReturnDirectionN() {
        Direction direction = Direction.getFromShortName('N');
        assertEquals(direction, Direction.NORTH);
    }

    public void whenGetFromShortNameWThenReturnDirectionW() {
        Direction direction = Direction.getFromShortName('W');
        assertEquals(direction, Direction.WEST);
    }

}
```

这个示例中，我们给DirectionSpec类指定了注解@Test，因此方法whenGetFromShortNameNThenReturnDirectionN和whenGetFromShortNameWThenReturnDirectionW都被视为测试。如果使用JUnit编写上述代码，需要给这两个方法都指定注解@Test。

4.3.2 注解@BeforeSuite、@BeforeTest、@BeforeGroups、@AfterGroups、@AfterTest和@AfterSuite

这4个注解都没有对应的JUnit注解。TestNG可使用XML配置将测试编组为套件。使用@BeforeSuite和@AfterSuite注解的方法分别在指定套件中的所有测试运行之前和之后运行。同样，使用@BeforeTest和@AfterTest注解的方法分别在测试类中的每个测试运行之前和之后运行。最后，TestNG测试还可组织为编组，而注解@BeforeGroups和@AfterGroups让你能够在指定编组中的所有测试运行之前和之后运行某些方法。

实现代码之后编写测试时，这些注解很有用，但在TDD中它们没有太大的用武之地。传统测试通常是作为一个独立的项目进行规划和编写的，而TDD要求我们每次编写一个测试，并确保一切都尽可能简单。最重要的是，单元测试必须能够快速运行，因此没有必要将它们分成套件或编组。测试的运行速度很快时，运行部分测试都是在浪费时间。例如，如果在15秒内能够运行所有测试，就没有必要运行部分测试。另一方面，如果测试的运行速度很慢，通常昭示着没有将外部依赖隔离。不管测试运行速度慢的原因是什么，都不能将运行部分测试作为解决方案，而应去修复问题。

另外，功能测试和集成测试通常运行速度更慢，必须以某种方式将测试分开。然而，最好在build.gradle中将它们分离，将每种测试作为一个独立的任务运行。

4.3.3 注解@BeforeClass和@AfterClass

这些注解在JUnit和TestNG中的作用相同：被注解的方法将分别在当前类中的所有测试运行之前和之后运行。唯一的差别是，TestNG不要求这些方法是静态的。原因是这两个框架运行测试方法的方式不同：JUnit运行每个测试时都使用不同的测试类实例，因此要让这些方法可重用，必

须将它们定义为静态的；而TestNG在同一个测试类实例中运行所有测试，因此不需要将这些方法定义为静态的。

4.3.4 注解`@BeforeMethod`和`@AfterMethod`

这些注解与JUnit注解`@Before`和`@After`等价，被注解的方法将在每个测试运行之前和之后运行。

4.3.5 注解参数`@Test(enable = false)`

JUnit和TestNG都能够禁用测试。为此，JUnit使用独立的注解`@Ignore`，而TestNG使用注解`@Test`的布尔参数`enable`。从功能上说，这两种做法的工作原理相同，唯一的差别是编写方式。

4.3.6 注解参数`@Test(expectedExceptions = SomeClass.class)`

在这个方面，JUnit占据了优势。虽然这两个框架提供的指定期望异常的方式相同（在JUnit中，参数为`expected`），但JUnit引入了更优雅的异常测试方式——规则（第2章使用过）。

4.3.7 TestNG 和 JUnit 差别小结

这两个框架还有很多其他的差别，但为简单起见，本书没有全部介绍。有关这方面的更详细信息，请参阅这两个框架的文档。

> 有关JUnit和TestNG的更详细信息，请参阅http://junit.org/和http://testng.org/。

TestNG提供的功能更多，也比JUnit更先进。本章将一直使用TestNG，你会对它有更深入的了解。需要指出的一点是，我们不会使用其任何高级功能，因为在TDD中编写单元测试时，很少需要用到。功能测试和集成测试与单元测试不同，可更好地演示TestNG的优越性；然而，你将在本书后面看到，有其他更适合编写这些测试的工具。

该使用哪个框架呢？如何选择由你决定，因为阅读完本章后，你将获得实际使用JUnit和TestNG的经验。

4.4 “遥控军舰”的需求

我们将完成著名编码套路Mars Rover的变种，这个编码套路最初是由Dallas Hack Club（http://dallashackclub.com/rover）发布的。

假设有一艘停留在海中的军舰，鉴于现在是21世纪，我们完全可以遥控它。

> 这里的任务是创建一个程序，让这艘军舰能够在海中游弋。

鉴于本书是一部介绍TDD的著作，而本章的主题为单元测试，所以我们将使用TDD方法开发一个应用程序，并将重点放在单元测试上。前一章学习了TDD理论，并获得了实际使用“红灯-绿灯-重构”过程的经验。这里将以此为基础，学习如何有效利用单元测试。具体地说，你将专注于当前要开发的单元，并学习如何隔离并忽略它可能使用的依赖。不仅如此，你还将尝试每次只专注于一个需求。有鉴于此，这里只提供粗略的需求：移动海中的遥控军舰。

为简化学习过程，所有的支持类都已编写好并进行了测试。这让你能够专注于手头的任务，同时确保这个练习简单明了。

4.5 开发“遥控军舰”

首先导入既有的Git仓库。

4.5.1 创建项目

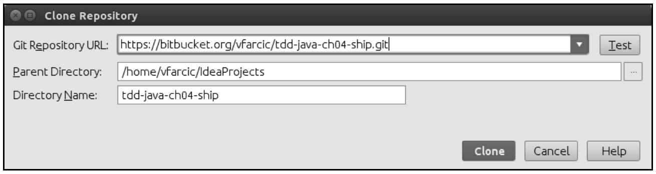
首先创建项目：

(1) 启动IntelliJ IDEA。如果打开了既有项目，请选择**File > Close Project**将其关闭。

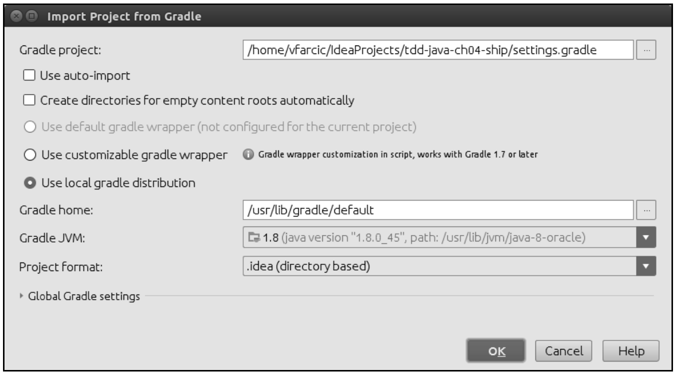
你将看到类似于下图的屏幕。



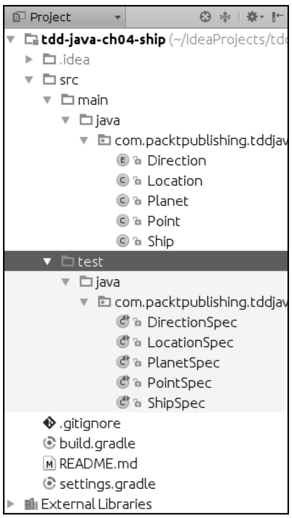
(2) 为从Git仓库导入项目，请单击**Check out from Version Control**并选择**Git**。然后在文本框**Git Repository URL**中输入https://bitbucket.org/vfarcic/tdd-java-ch04-ship.git，并单击按钮**Clone**。



(3) 被问及是否要打开这个项目时，回答**Yes**。接下来将出现对话框**Import Project from Gradle**，请单击按钮**OK**。



(4) IDEA将花些时间下载文件`build.gradle`中指定的依赖。下载完毕后，你将看到已经创建了一些类和相应的测试。



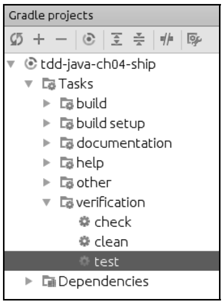
4.5.2　辅助类

假设这个项目最初是由你的一位同事开发的，他是位卓越的程序员和TDD践行者，你深信他编写的测试有极高的代码覆盖率。换言之，你完全可以依赖他已做的工作。然而，这位同事还未完成这个项目就去度假了，余下的工作将由你接手完成。他创建了所有辅助类：`Direction`、`Location`、`Planet`和`Point`。你注意到相应的测试类也已编写好，它们的名称与被测试的类相同，但包含后缀`Spec`（如`DirectionSpec`）。使用这个后缀旨在明确这样一点：它们不仅用于验证代码，还是可执行的规范。

除这些辅助类外，还有另外两个类：`Ship`（**实现**）和`ShipSpec`（**规范/测试**），你的大部分时间都将花在完善它们上。你将在`ShipSpec`中编写测试，再在`Ship`类中编写实现代码（与本书前面做的完全相同）。

我们知道，测试不仅提供了验证代码的途径，还是可执行的文档。因此从现在开始，我们将测试称为“规范”。

每次编写规范或实现规范的代码后，我们都将运行测试。从命令提示符执行命令`gradle test`，或使用IDEA工具窗口Gradle projects执行测试。



项目创建完毕后，着手处理第一个需求。

4.5.3　需求 1

要移动军舰，需要知道它当前的位置；另外，还需知道军舰面向哪个方向：北、南、东还是西。因此，第一个需求如下：

> 给定军舰的起始位置(**x, y**)以及它面向的方向（**N**、**S**、**E**或**W**）。

处理这个需求前，先看一下可使用的辅助类。`Point`类存储了坐标`x`和`y`，其构造函数如下：

```
public Point(int x, int y) {
```

```
        this.x = x;
        this.y = y;
    }
```

还有枚举类Direction，它定义的值如下：

```
public enum Direction {
    NORTH(0, 'N),
    EAST(1, 'E'),
    SOUTH(2, 'S'),
    WEST(3, 'W'),
    NONE(4, 'X');
}
```

最后，还有Location类，其构造函数将前述两个类的对象作为参数：

```
    public Location(Point point, Direction direction) {
        this.point = point;
        this.direction = direction;
    }
```

知道这些后，为第一个需求编写测试就非常容易。你应该像前一章那样做。

请尝试自己编写规范，完成后再将其与本书提供的解决方案进行比较。对于实现规范的代码，也这样做：尝试自己编写它们，完成后再与我们提供的解决方案进行比较。

1. 规范

这个需求的规范如下：

```
@Test
public class ShipSpec {

      public void whenInstantiatedThenLocationIsSet() {
        Location location = new Location(
        new Point(21, 13), Direction.NORTH);
        Ship ship = new Ship(location);
        assertEquals(ship.getLocation(), location);
    }
}
```

这个规范很简单，我们只做了这样的检查：传递给构造函数Ship的Location对象是否被存储；能否通过获取函数getLocation访问它。

> **注解@Test**
>
> 使用TestNG时，在类级指定注解@Test后，无需再指定应将哪些方法视为测试。在这里，所有的公有方法都被视为TestNG测试。

2. 实现

这个规范的实现非常简单，只需将构造函数的参数赋给变量location即可：

```
public class Ship {

    private final Location location;
    public Location getLocation() {
        return location;
    }

    public Ship(Location location) {
        this.location = location;
    }

}
```

完整的源代码可在仓库tdd-java-ch04-ship的分支req01-location（https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req01-location）中找到。

3. 重构

我们知道，需要为每个规范实例化Ship，因此需要重构规范类，在其中添加一个用@BeforeMethod注解的方法，如下所示：

```
@Test
public class ShipSpec {

    private Ship ship;
    private Location location;

    @BeforeMethod
        public void beforeTest() {
         Location location = new Location(
             new Point(21, 13), Direction.NORTH);
        ship = new Ship(location);
    }

    public void whenInstantiatedThenLocationIsSet() {
//        Location location = new Location(
//            new Point(21, 13), Direction.NORTH);
//        Ship ship = new Ship(location);
          assertEquals(ship.getLocation(), location);
    }
}
```

我们没有引入任何新的行为，而只将部分代码移到了用@BeforeMethod注解的方法中，以免编写后面的规范时重复这些代码。这样，运行每个测试时，都将使用location为参数实例化一个Ship对象。

4.5.4　需求 2

知道军舰在什么地方后，下面尝试移动它。首先，我们应该让它能够前进和后退。

> 实现让军舰前进和后退的命令（f和b）。

辅助类已包含方法`forward`和`backward`，它们实现了这项功能：

```
public boolean forward() {
    ...
}
```

1. 规范

在军舰朝北的情况下，如果我们向前移动它，结果将如何呢？其y坐标将减1。如果军舰面向东呢？其x坐标应加1。

面对这样的情况，你的第一反应应该是编写两个类似于下面的规范：

```
public void givenNorthWhenMoveForwardThenYDecreases() {
    ship.moveForward();
    assertEquals(ship.getLocation().getPoint().getY(), 12);
}

public void givenEastWhenMoveForwardThenXIncreases() {
    ship.getLocation().setDirection(Direction.EAST);
    ship.moveForward();
    assertEquals(ship.getLocation().getPoint().getX(), 22);
}
```

如果这样做，你至少还需编写两个规范，它们分别与军舰朝南和西相关。

然而，不应这样编写单元测试。大多数UT新手都会落入这样的陷阱，即指定方法的结果时，牵涉到它使用的方法、类和库的内部工作原理。这种做法在很多层面上都存在问题。

当前规范的单元中包含外部代码时，应考虑这样一点（至少在这里应该如此），即外部代码已经过测试。我们知道外部代码没有问题，因为每次修改代码后，我们都运行了所有测试。

> **每次修改实现代码后都再次运行所有测试。**
>
> 这确保对代码所做的修改不会带来任何意外的副作用。
>
> 每次修改实现代码后，都应运行所有测试。最理想的情况是，测试的执行速度很快，且开发人员能够在本地运行。将代码提交给版本控制系统后，应再次运行所有测试，确认代码合并没有带来任何问题。多位开发人员协作开发代码时，这显得尤其重要。你应使用Jenkins、Hudson、Travind、Bamboo和Go-CD等持续集成工具从仓库获取代码、对其进行编译并运行测试。

这种做法存在的另一个问题是，如果外部代码发生变化，将需要修改很多规范；而理想情况下，单元被修改时，应只需修改与之直接相关的规范。如果必须找出调用了该单元的所有地方，将既耗时又容易出错。

对于前面的需求，要为其编写规范，一种更容易、更快捷、更好的方式如下所示：

```
public void whenMoveForwardThenForward() {
    Location expected = location.copy();
    expected.forward();
    ship.moveForward();
    assertEquals(ship.getLocation(), expected);
}
```

由于Location已包含方法forward，因此只需确认正确调用了这个方法。为此，我们创建一个新的Location对象——expected，调用其方法forward，再将这个对象同调用方法moveForward后的军舰位置进行比较。

请注意，规范不仅用于验证代码，还被用作可执行的文档，最重要的是，它们还被用作思考和设计方式。前述修改后的规范更清晰地指出了其意图：应在Ship类中创建方法moveForward，并确保这个方法调用了location.forward。

2. 实现

有了这种简短而明确的规范后，编写实现代码应非常容易：

```
public boolean moveForward() {
    return location.forward();
}
```

3. 规范

前面规范并实现了前进功能，后退功能几乎完全相同：

```
public void whenMoveBackwardThenBackward() {
    Location expected = location.copy();
    expected.backward();
    ship.moveBackward();
    assertEquals(ship.getLocation(), expected);
}
```

4. 实现

与规范一样，后退功能的实现也很容易：

```
public boolean moveBackward() {
    return location.backward();
}
```

这个需求的完整源代码可在仓库tdd-java-ch04-ship的分支req02-forward-backward（https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req02-forward-backward）中找到。

4.5.5　需求 3

仅仅前后移动军舰意义不大，还应能够让军舰左转和右转。

> 实现让军舰左转和右转的命令（l和r）。

实现前一个需求后，这个需求实现起来应非常容易，因为其逻辑完全相同。辅助类`Location`已经包含实现这项需求的方法`turnLeft`和`turnRight`，我们只需将它们集成到`Ship`类中即可。

1. 规范

基于前面遵循的指导方针，可这样编写有关左转的规范：

```
public void whenTurnLeftThenLeft() {
    Location expected = location.copy();
    expected.turnLeft();
    ship.turnLeft();
    assertEquals(ship.getLocation(), expected);
}
```

2. 实现

你可以轻松编写让这个规范通过的代码：

```
public void turnLeft() {
    location.turnLeft();
}
```

3. 规范

右转应该与左转几乎完全相同：

```
public void whenTurnRightThenRight() {
    Location expected = location.copy();
    expected.turnRight();
    ship.turnRight();
    assertEquals(ship.getLocation(), expected);
}
```

4. 实现

最后结束这个需求，实现右转规范：

```
public void turnRight() {
    location.turnRight();
}
```

这个需求的完整源代码可在仓库`tdd-java-ch04-ship`的分支`req03-left-right`（https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req03-left-right）中找到。

4.5.6 需求 4

前面所做的一切都非常简单，因为有辅助类提供了所有功能。这个练习旨在让你学会不去测试最终结果，而专注于当前要开发的单元。这是为了加强信任，因为我们必须信任他人编写的代码（辅助类）。从这个需求开始，你将必须信任自己编写的代码。我们将继续前面的做法：编写规范、运行测试并发现它们以失败告终；再编写实现、运行测试并发现测试通过；最后，对我们认为存在改进空间的代码进行重构。同时，继续秉承这样的思路，即对单元（方法）进行测试时，不要过多考虑它将调用的方法或类。

实现各个命令（前进、后退、左转和右转）后，下面结合使用它们。我们应创建一个方法，它接受一个字符串作为参数，其中可包含任意数量的命令。每个命令都用一个字符表示：**f**表示前进、**b**表示后退、**l**表示左转、**r**表示右转。

> 军舰可接收一个包含命令的字符串（例如，`lrfb`相当于左转、右转、前进再后退）。

1. 规范

先看只包含字符`f`（前进）的命令参数：

```
public void whenReceiveCommandsFThenForward() {
    Location expected = location.copy();
    expected.forward();
    ship.receiveCommands("f");
    assertEquals(ship.getLocation(), expected);
}
```

这个规范几乎与`whenMoveForwardThenForward`相同，只是其中调用的是方法`ship.receiveCommands("f")method`。

2. 实现

应编写尽可能简单的代码，只要让规范能够通过即可，这样做的重要性如前所述。

> **应编写尽可能简单的代码，只要让测试能够通过即可。这可确保设计越来越清晰，并避免包含多余功能。**
>
> 这里的理念是，实现越简单，产品越好，维护也越容易。这种理念遵循了KISS原则，该原则指出，对大多数系统而言，保持简单而不是复杂化的效果最好。因此设计的主要目标是简约，必须避免不必要的复杂性。

现在正是应用这条规则的大好时机。你可能倾向于编写如下代码：

```
public void receiveCommands(String commands) {
```

```
        if (commands.charAt(0) == 'f') {
            moveForward();
        }
    }
```

这个代码示例中，你检查第一个字符是否是f，如果是就调用方法moveForward。可使用的其他变种还有很多，但如果坚持简约原则，下面的解决方案更好：

```
public void receiveCommands(String command) {
    moveForward();
}
```

这段代码能够让前述规范得以通过，且最简单、最简短。最后的代码可能与第一个版本更接近；随着情况越来越复杂，我们还可能使用循环或设计其他解决方案。但目前而言，我们每次专注于一个规范，并力图让代码尽可能简单。我们力图只专注于手头的任务，以理清思路。

为简洁起见，这里不再演示如何处理其他命令（b、l和r，你可自己规范并实现），而直接跳到这项需求中的最后一个规范。

3. 规范

能够处理单个命令（不管这个命令是什么）后，下面添加发送命令字符串的选项。相应规范如下：

```
public void whenReceiveCommandsThenAllAreExecuted() {
    Location expected = location.copy();
    expected.turnRight();
    expected.forward();
    expected.turnLeft();
    expected.backward();
    ship.receiveCommands("rflb");
    assertEquals(ship.getLocation(), expected);
}
```

这个规范有点长，但依然不太复杂。我们传递命令字符串rflb（右转、前进、左转和后退），并期望Location发生相应变化。与以前一样，我们没有验证最终结果（检查坐标是否发生相应变化），而检查是否正确调用了辅助类的方法。

4. 实现

实现代码可能如下所示：

```
public void receiveCommands(String commands) {
    for (char command : commands.toCharArray()) {
        switch(command) {
            case 'f':
                moveForward();
                break;
            case 'b':
```

```
                moveBackward();
                break;
            case 'l':
                turnLeft();
                break;
            case 'r':
                turnRight();
                break;
        }
    }
}
```

如果你尝试自己编写规范和实现，并遵循简约规则，很可能重构多次后才得到最终的解决方案。简约是关键，而重构通常是必须的。重构时别忘了，所有规范都必须在任何时候都能通过。

> **仅当所有测试都通过后才重构**
>
> 这样做的优点如下：重构是安全的。
>
> 如果所有可能受影响的实现代码都有测试，且所有测试都通过，那么重构将是相对安全的。大多数情况下不需要添加新的测试，而只需对既有测试做细微的修改即可。重构的预期结果是，在修改代码之前和之后，所有测试都能通过。

这项需求的完整源代码可在仓库`tdd-java-ch04-ship`的分支`req04-commands`（https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req04-commands）中找到。

4.5.7　需求 5

与其他星球一样，地球也是圆的。用地图表示地球时，到达地图的边缘后，将进入另一边。例如，如果我们向东移动到太平洋的最远端，将到达地图的西边，并离美洲越来越近。另外，为让移动更容易，可将地图定义为网格。这个网格有长度和高度，分别对应于X轴和Y轴；同时，这个网格有最大的X值和Y值。

> 实现从网格的一边转到另一边。

1. 规范

首先，我们可以将X坐标和Y坐标为最大值的`Planet`对象传递给`Ship`构造函数。所幸`Planet`也是一个辅助类，已经创建好并经过了测试。我们只需实例化这个类，并将得到的对象传递给`Ship`构造函数：

```
public void whenInstantiatedThenPlanetIsStored() {
    Point max = new Point(50, 50);
    Planet planet = new Planet(max);
```

```
        ship = new Ship(location, planet);
        assertEquals(ship.getPlanet(), planet);
    }
```

我们将星球的大小定义为50 × 50，并使用这种大小实例化Planet类，再将生成的对象传递给Ship构造函数。你可能注意到，这个构造函数接受一个额外的测试，而当前它只接受一个Location对象作为参数。为实现这个规范，必须让这个构造函数同时将一个Planet对象作为参数。

如何实现这个规范，同时又不破坏任何既有规范呢?

2. 实现

我们采用自下而上的方法。规范末尾的断言要求有一个planet获取方法：

```
    private Planet planet;
    public Planet getPlanet() {
        return planet;
    }
```

接下来，需要让构造函数将一个Planet对象作为第二个参数，并将其赋给刚才添加的变量planet。你首先想到的可能是在既有的构造函数中添加这个参数，但这将破坏很多既有规范，因为它们使用只有一个参数的构造函数。因此我们别无选择，只能再添加一个构造函数：

```
    public Ship(Location location) {
        this.location = location;
    }
    public Ship(Location location, Planet planet) {
        this.location = location;
        this.planet = planet;
    }
```

运行所有规范（测试），并确认它们都通过。

3. 重构

基于我们制定的规范，必须创建第二个构造函数。因为如果修改既有的构造函数，将破坏既有测试。然而，鉴于现在所有测试都通过，我们可以做些重构，以便能够将只接受一个参数的构造函数删除。规范类已包含一个beforeTest方法，它将在每个测试之前运行。对于刚才制定的规范，我们可以将其中除断言外的其他所有代码都移到这个beforeTest方法中：

```
public class ShipSpec {
...
    private Planet planet;

    @BeforeMethod
    public void beforeTest() {
        Point max = new Point(50, 50);
        location = new Location(new Point(21, 13), Direction.NORTH);
```

```
        planet = new Planet(max);
//        ship = new Ship(location);
        ship = new Ship(location, planet);
    }
    public void whenInstantiatedThenPlanetIsStored() {
//        Point max = new Point(50, 50);
//        Planet planet = new Planet(max);
//        ship = new Ship(location, planet);
        assertEquals(ship.getPlanet(), planet);
    }
}
```

这样的修改后，将不再使用只接受一个参数的Ship构造函数。通过运行所有规范可确认，这种修改是可行的。

现在，我们没有使用接受一个参数的构造函数，因此可将其从实现类中删除：

```
public class Ship {
...
//    public Ship(Location location) {
//        this.location = location;
//    }
    public Ship(Location location, Planet planet) {
        this.location = location;
        this.planet = planet;
    }
...
}
```

这样重构后，所有规范都通过了。这种重构没有改变任何既有功能，也没有破坏任何东西，且整个重构过程很快就完成了。

下面处理回转本身。

4. 规范

与前面一样，辅助类提供了我们需要的所有功能。前面使用的都是不接受任何参数的方法`Location.forward`，但为实现回转，必须使用该方法的另一个重载版本——`Location.forward(Point max)`，它在到达网格边缘时将军舰转到另一边。前一个规范中，我们确保使用`Point`对象`max`实例化一个`Planet`对象，并将其传递给`Ship`构造函数；而这里的任务是，确保前进时根据`max`确定当前位置。为此，可这样编写这个规范：

```
/* 考虑到本书版式，我们将这个方法的名称缩短了。这个测试旨在检查军舰跨越网格右边缘后的行为。*/
public void overpassEastBoundary() {
    location.setDirection(Direction.EAST);
    location.getPoint().setX(planet.getMax().getX());
    ship.receiveCommands("f");
    assertEquals(location.getX(), 1);
}
```

5. 实现

至此，你应该习惯了每次专注于一个单元，并信任之前实现的单元都像期望的那样工作。这个实现没什么不同，我们只需确保调用方法location.forward时使用了最大坐标：

```
    public boolean moveForward() {
//        return location.forward();
        return location.forward(planet.getMax());
    }
```

对于方法backward，规范和实现与此相同。为简洁起见，本书没有列出这些代码，但你可在源代码中找到它们。

这个需求的完整源代码可在仓库tdd-java-ch04-ship的分支req05-wrap（https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req05-wrap）中找到。

4.5.8　需求 6

我们就要完成了，这是最后一项需求。

虽然地球表面的很大一部分（大约70%）都被水覆盖，但还有一些大洲和岛屿，它们对遥控军舰来说就是障碍。我们需要能够检测接着移动军舰是否会撞上这些障碍，如果会，就应放弃这样的移动，让军舰留在原地并报告将遇到的障碍。

> 每次移动前都进行障碍检测。如果执行指定的命令将遇到障碍，军舰应放弃移动，留在原地并报告遇到的障碍。

这个规范及其实现与前面所做的很像，这些工作留给你自己去完成。

下面几个小提示可能会对你有所帮助：

- Planet类有一个构造函数将障碍列表作为参数。每个障碍都是一个Point实例。
- 方法Location.forward和Location.backward都有将障碍列表作为参数的重载版本，它们在移动成功时返回true，在移动失败时返回false。你可使用这个返回的布尔值创建方法Ship.receiveCommands所需的状态报告。
- 方法receiveCommands应返回一个字符串，以指出每个命令的状态。例如，你可使用O表示OK，用X表示失败（例如，OOXO表示OK、OK、失败和OK）。

这个需求的完整源代码可在仓库tdd-java-ch04-ship的分支req06-obstacles（https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req06-obstacles）中找到。

4.6 小结

本章中，我们使用了测试框架TestNG。与使用JNnit时相比，没有太多不同，因为我们没有使用TestNG的任何高级功能，如数据提供器、工厂等。在TDD中，我们可能根本不需要使用这些高级功能。

请访问http://testng.org/，对TestNG进行探索，再决定哪个框架最能满足你的需求。

本章的主要目标是介绍如何每次专注于一个单元。我们编写了大量辅助类，并尽量忽略其内部工作原理。很多情况下，我们编写规范时都没有验证最终结果是否正确，而检查待实现的方法是否调用了辅助类的正确方法。实际工作中，你将与其他小组成员协作开发项目，因此学会如何专注于分配给你的任务并相信他人开发的代码像预期的那样工作至关重要。对于第三方库，也应采取同样的态度。如果对我们调用的内部处理可能出现的情况都进行测试，代价实在太高，况且还有其他类型的测试会尝试覆盖这些可能性。进行单元测试时，应专注于当前的单元。

至此，你对在TDD中如何有效利用单元测试有更深入的认识，下面该深入探索TDD提供的其他优点了。具体地说，我们将探索如何将应用程序设计得更好。

第 5 章 设计——难以测试说明设计不佳

“大道至简。”

——列奥纳多 · 达 · 芬奇

以前，软件行业都专注于快速开发，心里只有成本和时间，质量只是次要目标。因为开发人员存在误区，认为客户对此漠不关心。

现在，随着各种平台和设备的联系日益紧密，质量成了客户需求的重中之重。卓越的应用程序都提供卓越的服务且响应时间合理，即便大量用户同时发出大量请求亦是如此。

质量卓越的应用程序都设计良好，而良好的设计意味着可伸缩性、安全性、可维护性和众多其他优良品质。

本章以传统方法和TDD方法开发同一款应用程序，以此探索TDD如何引导开发人员走向通往良好设计和最佳实践的道路。

本章涵盖如下主题：

- 为何要关心设计；
- 设计方面的考量；
- 传统的开发流程；
- TDD方法；
- Hamcrest。

5.1 为何要关心设计

无论你是专家还是初学者，都会在编码领域遇到看起来很怪异的代码，进而感觉有什么地方不对。偶尔甚至会心生疑惑，前一位程序员为何要以如此糟糕的方式实现方法或类？这是因为每

种功能都有很多不同的实现方式，而每种方式都是独一无二的。在如此之多的实现方式中，到底哪种是最佳的呢？答案是，不管白猫还是黑猫，能抓到老鼠的就是好猫。然而，寻找更佳的解决方案时，确实有一些因素需要考虑。此时，设计就显得特别重要。

设计原则

TDD倡导程序员遵守一些让代码更清晰、更易读的原则和良好实践，从而确保代码易于理解并能安全地修改。下面看一些基本的软件设计原则。

1. 你不会需要它

YAGNI是设计原则You Ain't Gonna Need It（你不会需要它）的首字母缩写，旨在消除所有冗余代码，并专注于当前而不是未来的功能。代码越少，需要维护的代码就越少，同时引入bug的可能性也越小。

有关YAGNI的更详细信息，请参阅Martin Fowler撰写的相关文章，网址为http://martinfowler.com/bliki/Yagni.html。

2. 不要自我重复

不要自我重复（DRY）原则基于的理念是，重用而不是复制以前编写的代码。这样做的好处是，需要维护的代码更少，并确信使用的代码是可行的——这是天大的好事。另外，这还有助于在代码中发现新的抽象层级。

有关这个设计原则的更详细信息，请参阅http://en.wikipedia.org/wiki/Don%27t_repeat_yourself。

3. 保持简单

这个原则是Kelly Johnson提出的，其首字母缩写令人迷惑。这个原则指出，越简单的东西越能实现其功能。

有关这条原则背后的故事，请参阅http://en.wikipedia.org/wiki/KISS_principle。

4. 奥卡姆剃刀原理

奥卡姆剃刀原理是一个哲学原则，而非软件工程原则，但它依然适用于我们从事的工作。这个原则与前一个原则极其相似，其主要推论如下：

> "如果你有两个或多个类似的解决方案，选择最简单的。"
>
> ——奥卡姆的威廉

有关奥卡姆剃刀原理的详细信息，请参阅http://en.wikipedia.org/wiki/Occam%27s_razor。

5

5. SOLID

SOLID是Robert C. Martin发明的一个首字母缩写，涵盖5个面向对象编程的基本原则。通过遵守这5个原则，开发人员更有可能打造卓越、持久和易于维护的应用程序：

- **单一职责原则**：一个类应该只有一个导致它需要修改的原因。
- **开–闭原则**：类应该对扩展是开放的，对修改是封闭的。这个原则最初由Bertrand Meyer提出。
- **里氏替换原则**：这个原则是Barbara Liskov提出的，她指出，类应该能够被扩展它的类替换。
- **接口分离原则**：提供多个具体接口胜过提供单个通用接口。
- **依赖倒转原则**：类应依赖于抽象而不是实现，这意味着类依赖必须专注于做什么而不是如何做。

有关SOLID和其他相关原则的更详细信息，请参阅http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod。

这里的前4个原则是TDD思维的核心，因为它们旨在简化代码，而最后一个原则专注于类的编写和依赖关系。

无论在测试驱动开发还是非测试驱动开发中，这些原则都适用也必须遵守，因为它们让代码更容易维护，还将带来其他好处。有关如何正确应用这些原则可写部专著，这里没有时间探讨，但建议你对它们做更深入的研究。

在本章中，你会看到TDD将引导开发人员毫不费力地应用这里介绍的一些原则。我们将使用TDD和非TDD方法实现著名游戏**Connect4**的小型版本，但麻雀虽小，五脏俱全。请注意，Gradle项目创建等重复部分与本章目标无关，故省略。

5.2　Connect4

Connect4是一款流行的桌面游戏，其规则少而简单，玩起来很容易。

> Connect4是一款两人玩的连接游戏。玩家首先选择颜色，然后轮流将碟片放入7列6行的网格中。碟片垂直下落，停留在当前列中下一个未占据的位置。玩家的目标是抢在对手前将自己的4个相邻碟片排成水平线、垂直线或对角线。
>
> 有关这款游戏的更详细信息，请参阅维基百科（http://en.wikipedia.org/wiki/Connect_Four）。

需求

为编写Connect4的两个实现，下面以需求的方式转录这个游戏的规则。这两个开发过程都以这些需求为起点，完成后我们将对代码做些解释，并对这两个实现进行比较：

(1) 棋盘为7列6行，所有格子都是空的。

(2) 玩家从列顶放入碟片。如果整列为空，放入的碟片将落到底部。在特定列中，后放入的碟片将叠在前面放入的碟片之上。

(3) 这是一款两人玩的游戏，每位玩家的碟片用一种颜色表示：一位玩家为红色（'R'），另一位玩家为绿色（'G'）。玩家轮流放入碟片，每次放入一个。

(4) 我们要在玩家放入碟片或发生错误时提供反馈：每当玩家放入碟片后，都使用输出指出棋盘状态。

(5) 无法再放入碟片时游戏结束，结果为平局。

(6) 玩家放入碟片后，如果将其3个以上碟片连成垂直线，该玩家将获胜。

(7) 玩家放入碟片后，如果将其3个以上碟片连成水平线，该玩家将获胜。

(8) 玩家放入碟片后，如果将其3个以上碟片连成对角线，该玩家将获胜。

5.3 完成 Connect4 实现后再测试

这是传统的开发方法，专注于解决问题而不是测试。有些人和公司对自动化测试的价值置若罔闻，而依赖用户执行**用户验收测试**。

这种用户验收测试在受控环境（最好与生产环境完全相同）中重现真实场景，让用户执行大量不同的任务，以验证应用程序的正确性。如果有操作失败，代码就是不可接受的，因为它们破坏了某些功能，或不像预期的那样工作。

另外，很多公司还将单元测试作为一种执行早期回归检查的手段。这些单元测试是在开发结束后编写的，旨在覆盖尽可能多的代码。最后，执行代码覆盖分析，以确定这些单元测试覆盖了哪些代码；代码覆盖率越高，说明交付的软件质量越好。

下面使用这种方法实现Connect4。每个需求都列出了与之相关的代码。这些代码不是以增量方式编写的，因此有些代码片段可能包含与当前需求无关的代码行。

5.3.1 需求 1

下面先看第一个需求。

> 棋盘为7列6行，且整个棋盘都是空的。

这个需求的实现非常简单。我们只需定义“空”的表示方式，并创建存储游戏数据的数据结构即可。注意，这里还定义了玩家使用的颜色：

```
public class Connect4 {
    public enum Color {
        RED('R'), GREEN('G'), EMPTY(' ');

        private final char value;

        Color(char value) { this.value = value; }

        @Override
        public String toString() {
            return String.valueOf(value);
        }
    }

public static final int COLUMNS = 7;

public static final int ROWS = 6;

private Color[][] board = new Color[COLUMNS][ROWS];

public Connect4() {
    for (Color[] column : board) {
        Arrays.fill(column, Color.EMPTY);
    }
  }
}
```

5.3.2　需求 2

第二个需求开始实现游戏逻辑。

> 玩家从列顶放入碟片。如果整列都是空的，放入的碟片将落到底部。在特定列中，后放入的碟片将叠在前面放入的碟片之上。

这部分需要考虑棋盘的边界，还需标出哪些位置被占据（使用`Color.RED`指出）。最后，我们创建了第一个私有方法，这是一个辅助方法，计算在给定列放入多少个碟片：

```
public void putDisc(int column) {
    if (column > 0 && column <= COLUMNS) {
        int numOfDiscs =
        getNumberOfDiscsInColumn(column - 1);
        if (numOfDiscs < ROWS) {
            board[column - 1][numOfDiscs] =
            Color.RED;
        }
    }
```

```
}

private int getNumberOfDiscsInColumn(int column) {
    if (column >= 0 && column < COLUMNS) {
      int row;
      for (row = 0; row < ROWS; row++) {
        if (Color.EMPTY == board[column][row]) {
          return row;
        }
      }
      return row;
    }
    return -1;
}
```

5.3.3　需求 3

这个需求引入了更多游戏逻辑。

> 这是一款两人玩的游戏，每位玩家的碟片用一种颜色表示：一位玩家为红色（'R'），另一位玩家为绿色（'G'）。玩家轮流放入碟片，每次放入一个。

我们需要保存当前玩家，以判断接下来轮到哪个玩家。还需要一个切换玩家的函数，以实现轮流逻辑。函数putDisc中，添加一些与该需求相关的代码。具体地说，将相应的棋盘位置分配给当前玩家，并根据游戏规则切换玩家：

```
...
private Color currentPlayer = Color.RED;

private void switchPlayer() {
    if (Color.RED == currentPlayer)
        currentPlayer = Color.GREEN;
    } else {
        currentPlayer = Color.RED;
    }
}

public void putDisc(int column) {
    if (column > 0 && column <= COLUMNS) {
        int numOfDiscs =
        getNumberOfDiscsInColumn(column - 1);
        if (numOfDiscs < ROWS) {
            board[column - 1][numOfDiscs] =
             currentPlayer;
            switchPlayer();
        }
    }
}
...
```

5.3.4　需求 4

为使用户知道游戏当前状态，需要添加一些输出。

> 我们要在玩家放入碟片或发生错误时提供反馈：每当玩家放入碟片后，都使用输出指出棋盘状态。

需求没有指定输出通道。出于简化考虑，我们决定使用系统标准输出以指出发生的事件。执行操作的每个方法都添加了几行提供输出的代码，让用户知道游戏的当前状态：

```
...
private static final String DELIMITER = "|";

private void switchPlayer() {
    if (Color.RED == currentPlayer) {
        currentPlayer = Color.GREEN;
    } else {
        currentPlayer = Color.RED;
    }
    System.out.println("Current turn: " +
         currentPlayer);
}

public void printBoard() {
    for (int row = ROWS - 1; row >= 0; --row) {
        StringJoiner stringJoiner =
            new StringJoiner(DELIMITER,
                    DELIMITER,
                    DELIMITER);
        for (int col = 0; col < COLUMNS; ++col) {
            stringJoiner
            .add(board[col][row].toString());
        }
        System.out.println(
        stringJoiner.toString());
    }
}

public void putDisc(int column) {
    if (column > 0 && column <= COLUMNS) {
        int numOfDiscs =
        getNumberOfDiscsInColumn(column - 1);
        if (numOfDiscs < ROWS) {
            board[column - 1][numOfDiscs] =
            currentPlayer;
            printBoard();
            switchPlayer();
        } else {
            System.out.println(numOfDiscs);
            System.out.println("There's no room " +
```

```
                "for a new disc in this column");
            printBoard();
        }
    } else {
        System.out.println("Column out of bounds");
        printBoard();
    }
}
...
```

5.3.5 需求 5

第一个游戏结束条件。

> 无法再放入碟片时游戏结束，结果为平局。

下面的代码是一种可能的实现：

```
...
public boolean isFinished() {
    int numOfDiscs = 0;
    for (int col = 0; col < COLUMNS; ++col) {
        numOfDiscs +=
        getNumberOfDiscsInColumn(col);
    }
    if (numOfDiscs >= COLUMNS * ROWS) {
        System.out.println("It's a draw");
        return true;
    }
    return false;
}
...
```


5.3.6 需求 6

第一个获胜条件。

> 玩家放入碟片后，如果将其3个以上碟片连成垂直线，该玩家将获胜。

私有方法`checkWinCondition`实现了这条规则，它检查最后放入的碟片会否让玩家获胜：

```
...
private Color winner;

public static final int DISCS_FOR_WIN = 4;
```

```
public void putDisc(int column) {
        ...
        if (numOfDiscs < ROWS) {
            board[column - 1][numOfDiscs] =
            currentPlayer;
            printBoard();
            checkWinCondition(column - 1,
            numOfDiscs);
            switchPlayer();
      ...
}

private void checkWinCondition(int col, int row) {
    Pattern winPattern =
        Pattern.compile(".*" + currentPlayer +
        "{" + DISCS_FOR_WIN + "}.*");

    // 检查垂直方向
    StringJoiner stringJoiner =
        new StringJoiner("");
    for (int auxRow = 0; auxRow < ROWS; ++auxRow) {
        stringJoiner
        .add(board[col][auxRow].toString());
    }
    if (winPattern.matcher(stringJoiner.toString())
            .matches()) {
        winner = currentPlayer;
        System.out.println(currentPlayer +
        " wins");
    }
}

public boolean isFinished() {
    if (winner != null) return true;
    ...
}
...
```

5.3.7　需求 7

获胜条件与前面相同，但方向不同。

> 玩家放入碟片后，如果将其3个以上碟片连成水平线，该玩家将获胜。

实现这条规则只需几行代码，如下所示：

```
...
private void checkWinCondition(int col, int row) {
    ...
    // 检查水平方向
```

```
        stringJoiner = new StringJoiner("");
        for (int column = 0; column < COLUMNS;
                ++column) {
            stringJoiner
            .add(board[column][row].toString());
        }
        if (winPattern.matcher(stringJoiner.toString())
            .matches()) {
            winner = currentPlayer;
            System.out.println(currentPlayer +
             " wins");
            return;
        }
        ...
    }
    ...
```

5.3.8 需求 8

这是最后一个获胜条件，它与前两个获胜条件很像，但为对角线方向。

> 玩家放入碟片后，如果将其3个以上碟片连成对角线，该玩家将获胜。

下面是这个需求的一种可能的实现，代码与其他两个获胜条件很像，因为需要满足的条件类似：

```
    ...
    private void checkWinCondition(int col, int row) {
        ...
        // 检查对角线方向
        int startOffset = Math.min(col, row);
        int column = col - startOffset,
            auxRow = row - startOffset;
        stringJoiner = new StringJoiner("");
        do {
            stringJoiner
             .add(board[column++][auxRow++].toString());
        } while (column < COLUMNS && auxRow < ROWS);

         if (winPattern.matcher(stringJoiner.toString())
            .matches()) {
            winner = currentPlayer;
            System.out.println(currentPlayer +
            " wins");
            return;
        }

        startOffset = Math.min(col, ROWS - 1 - row);
        column = col - startOffset;
```

```
        auxRow = row + startOffset;
        stringJoiner = new StringJoiner("");
        do {
            stringJoiner
             .add(board[column++][auxRow--].toString());
        } while (column < COLUMNS && auxRow >= 0);

        if (winPattern.matcher(stringJoiner.toString())
            .matches()) {
            winner = currentPlayer;
            System.out.println(currentPlayer +
            " wins");
        }
    }
    ...
```

至此，我们创建了一个类，它包含一个构造函数、3个公有方法和3个私有方法，应用程序的逻辑分散在这些方法中。该实现最大的缺陷在于，这个类很难维护，因为诸如`checkWinCondition`等重要方法都不简单，以后修改它们时很容易引入bug。

如果要查看完整的源代码，可在仓库https://bitbucket.org/vfarcic/tdd-java-ch05-design.git中找到。

这个小型示例旨在演示传统开发方法存在的常见问题，要演示SOLID原则等主题，需要使用更大的项目。

包含数百个类的大型项目中，如果出现问题，开发人员需要花很多时间，以类似于外科手术的方式解决。开发人员的很大一部分时间都花在研究复杂代码及其工作原理上，而无法将这些时间用于开发新功能。

5.4　使用 TDD 实现 Connect4

至此，我们知道了TDD的工作原理：先编写测试，再编写实现，然后进行重构。我们将使用这个过程处理每个需求，但只列出最终结果，至于具体的“红灯-绿灯-重构”过程请自行思考。为让事情变得更有趣，我们将尽可能在测试中使用Hamcrest框架。

5.4.1　Hamcrest

第2章说过，Hamcrest可提高测试的可读性。它使用匹配器降低复杂度，让断言的语义更明确、更容易理解。测试失败时，通过解读断言中使用的匹配器，更容易理解显示的错误。另外，开发人员还可添加测试失败时显示的消息。

`Hamcrest`库充斥着各种用于不同对象类型和集合的匹配器。下面编写代码并体会Hamcrest吧。

5.4.2　需求 1

首先从第一个需求着手。

> 棋盘为7列6行，所有格子都是空的。

这个需求一点都不难。它指定了棋盘边界，但没有描述任何行为，因此只需确保游戏开始时棋盘为空即可。这意味着游戏开始时碟片数为零。然而，后面必须考虑这个需求。

1. 测试

下面是这个需求的测试类。其中有一个实例化受测类的方法，这个方法使得每个测试都将使用全新的受测对象；还有一个测试，它验证游戏开始时没有任何碟片，即整个棋盘空空如也：

```
public class Connect4TDDSpec {

    private Connect4TDD tested;

    @Before
    public void beforeEachTest() {
        tested = new Connect4TDD();
    }

    @Test
    public void whenTheGameIsStartedTheBoardIsEmpty() {
        assertThat(tested.getNumberOfDiscs(), is(0));
    }

}
```


2. 代码

下面是前一个规范的TDD实现。看看这个针对第一项需求的解决方案有多简洁——一个通过单行代码返回结果的简单方法：

```
public class Connect4TDD {

    public int getNumberOfDiscs() {
        return 0;
    }

}
```

5.4.3　需求 2

下面实现第二个需求。

> 玩家从列顶放入碟片。如果整列都是空的，放入的碟片将落到底部。在特定列中，后放入的碟片将叠在前面放入的碟片之上。

可将这个需求分解为如下测试：

- 碟片被加入空列时，其位置为0；
- 碟片被加入已经有一个碟片的列时，其位置为1；
- 每加入一个碟片，总碟片数都加1；
- 如果碟片位于棋盘边界外，将引发运行阶段异常；
- 向已满的列中加入碟片时，将引发运行阶段异常。

其中，最后两个测试是从第一项需求衍生而来的，它们与棋盘边界或棋盘行为相关。

1. 测试

前述测试的Java实现如下：

```
@Test
public void
whenDiscOutsideBoardThenRuntimeException() {
    int column = -1;
    exception.expect(RuntimeException.class);
    exception.expectMessage("Invalid column " +
         column);
    tested.putDiscInColumn(column);
}

@Test
public void
whenFirstDiscInsertedInColumnThenPositionIsZero() {
    int column = 1;
    assertThat(tested.putDiscInColumn(column),
            is(0));
}

@Test
public void
whenSecondDiscInsertedInColumnThenPositionIsOne() {
        int column = 1;
        tested.putDiscInColumn(column);
        assertThat(tested.putDiscInColumn(column),
        is(1));
    }

@Test
public void
whenDiscInsertedThenNumberOfDiscsIncreases() {
    int column = 1;
```

```
    tested.putDiscInColumn(column);
    assertThat(tested.getNumberOfDiscs(), is(1));
}

@Test
public void
whenNoMoreRoomInColumnThenRuntimeException() {
    int column = 1;
    int maxDiscsInColumn = 6; // the number of rows
    for (int times = 0;
        times < maxDiscsInColumn;
        ++times) {
        tested.putDiscInColumn(column);
    }
    exception.expect(RuntimeException.class);
    exception
        .expectMessage("No more room in column " +
             column);
    tested.putDiscInColumn(column);
}
```

2. 代码

下面是让这些测试得以通过的代码：

```
private static final int ROWS = 6;

private static final int COLUMNS = 7;

private static final String EMPTY = " ";

private String[][] board =
    new String[ROWS][COLUMNS];

public Connect4TDD() {
    for (String[] row : board)
        Arrays.fill(row, EMPTY);
}

public int getNumberOfDiscs() {
    return IntStream.range(0, COLUMNS)
            .map(this::getNumberOfDiscsInColumn).sum();
}

private int getNumberOfDiscsInColumn(int column) {
    return (int) IntStream.range(0, ROWS)
            .filter(row -> !EMPTY
            .equals(board[row][column]))
            .count();
}

public int putDiscInColumn(int column) {
    checkColumn(column);
```

```
        int row = getNumberOfDiscsInColumn(column);
        checkPositionToInsert(row, column);
        board[row][column] = "X";
        return row;
    }

    private void checkColumn(int column) {
        if (column < 0 || column >= COLUMNS)
            throw new RuntimeException(
            "Invalid column " + column);
    }

    private void
    checkPositionToInsert(int row, int column) {
        if (row == ROWS)
          throw new RuntimeException(
            "No more room in column " + column);
    }
```

5.4.4 需求 3

第三个需求规范了游戏逻辑。

> 这是一款两人玩的游戏，每位玩家的碟片用一种颜色表示：一位玩家为红色（'R'），另一位玩家为绿色（'G'）。玩家轮流放入碟片，每次放入一个。

1. 测试

下面的测试验证新增的功能。为简洁起见，假定总是红方先来：

```
@Test
public void
whenFirstPlayerPlaysThenDiscColorIsRed() {
    assertThat(tested.getCurrentPlayer(), is("R"));
}

@Test
public void
whenSecondPlayerPlaysThenDiscColorIsRed() {
    int column = 1;
    tested.putDiscInColumn(column);
    assertThat(tested.getCurrentPlayer(), is("G"));
}
```

2. 代码

为实现这项功能，需要创建两个方法。方法`putDiscInColumn`中，返回碟片所在行前调用方法`switchPlayer`：

```
private static final String RED = "R";
```

```
private static final String GREEN = "G";

private String currentPlayer = RED;

public Connect4TDD() {
    for (String[] row : board)
        Arrays.fill(row, EMPTY);
}

public String getCurrentPlayer() {
    return currentPlayer;
}

private void switchPlayer() {
    if (RED.equals(currentPlayer))
        currentPlayer = GREEN;
    else currentPlayer = RED;
}

public int putDiscInColumn(int column) {
    ...
    switchPlayer();
    return row;
}
```


5.4.5　需求 4

接下来，需要让玩家知道游戏的当前状态。

> 我们要在玩家放入碟片或发生错误时提供反馈：每当玩家放入碟片后，都使用输出指出棋盘的状态。

1. 测试

发生错误时应引发异常，但前面的测试已覆盖这一点，因此这里只需编写两个测试。另外，为提高可测试性，我们需要在构造函数中新增一个参数。通过引入这个参数，输出测试将更容易：

```
private OutputStream output;

@Before
public void beforeEachTest() {
    output = new ByteArrayOutputStream();
    tested = new Connect4TDD(
          new PrintStream(output));
}

@Test
public void
```

```
whenAskedForCurrentPlayerTheOutputNotice() {
    tested.getCurrentPlayer();
    assertThat(output.toString(),
          containsString("Player R turn"));
}

@Test
public void
whenADiscIsIntroducedTheBoardIsPrinted() {
    int column = 1;
    tested.putDiscInColumn(column);
    assertThat(output.toString(),
        containsString("| |R| | | | | |"));
}
```

2. 代码

下面是让前述测试通过的一种可能实现。Connect4TDD类的构造函数现在有一个参数，多个方法中都使用了这个参数以输出有关事件或操作的描述：

```
private static final String DELIMITER = "|";

public Connect4TDD(PrintStream out) {
    outputChannel = out;
    for (String[] row : board)
        Arrays.fill(row, EMPTY);
}

public String getCurrentPlayer() {
    outputChannel.printf("Player %s turn%n",
        currentPlayer);
    return currentPlayer;
}

private void printBoard() {
    for (int row = ROWS - 1; row >= 0; row--) {
        StringJoiner stringJoiner =
        new StringJoiner(DELIMITER,
                     DELIMITER,
                     DELIMITER);
        Stream.of(board[row])
        .forEachOrdered(stringJoiner::add);
        outputChannel
        .println(stringJoiner.toString());
    }
}

public int putDiscInColumn(int column) {
    ...
    printBoard();
    switchPlayer();
    return row;
}
```

5.4.6　需求 5

这个需求告诉系统游戏是否结束。

> 无法再放入碟片时游戏结束，结果为平局。

1. 测试

需要测试的条件有两个，一是游戏刚开始时肯定未结束，二是棋盘已满时必须结束：

```
@Test
public void whenTheGameStartsItIsNotFinished() {
    assertFalse("The game must not be finished",
        tested.isFinished());
}

@Test
public void
whenNoDiscCanBeIntroducedTheGamesIsFinished() {
    for (int row = 0; row < 6; row++)
        for (int column = 0; column < 7; column++)
            tested.putDiscInColumn(column);
    assertTrue("The game must be finished",
        tested.isFinished());
}
```


2. 代码

下面是这两个测试的简单解决方案：

```
public boolean isFinished() {
    return getNumberOfDiscs() == ROWS * COLUMNS;
}
```

5.4.7　需求 6

这是第一个获胜条件。

> 玩家放入碟片后，如果将其3个以上碟片连成垂直线，该玩家将获胜。

1. 测试

事实上，只需做一项检查——如果当前加入的碟片与其他3个碟片连成一条水平线，则当前玩家将获胜：

```
@Test
public void
```

```
when4VerticalDiscsAreConnectedThenPlayerWins() {
    for (int row = 0; row < 3; row++) {
        tested.putDiscInColumn(1); // R
        tested.putDiscInColumn(2); // G
    }
    assertThat(tested.getWinner(),
        isEmptyString());
    tested.putDiscInColumn(1); // R
    assertThat(tested.getWinner(), is("R"));
}
```

2. 代码

对方法putDiscInColumn做了两个修改。还创建了一个新方法——checkWinner：

```
private static final int DISCS_TO_WIN = 4;

private String winner = "";

private void checkWinner(int row, int column) {
    if (winner.isEmpty()) {
        String colour = board[row][column];
        Pattern winPattern =
        Pattern.compile(".*" + colour + "{" +
          DISCS_TO_WIN + "}.*");

        String vertical = IntStream.range(0, ROWS)
                .mapToObj(r -> board[r][column])
                .reduce(String::concat).get();
        if (winPattern.matcher(vertical).matches())
            winner = colour;
    }
}
```

5.4.8　需求 7

这是第二个获胜条件，与前一个获胜条件很像。

> 玩家放入碟片后，如果将其3个以上碟片连成水平线，该玩家将获胜。

1. 测试

这次我们尝试在相邻的列中插入碟片以获胜：

```
@Test
public void
when4HorizontalDiscsAreConnectedThenPlayerWins() {
    int column;
    for (column = 0; column < 3; column++) {
```

```
        tested.putDiscInColumn(column); // R
        tested.putDiscInColumn(column); // G
    }
    assertThat(tested.getWinner(),
        isEmptyString());
    tested.putDiscInColumn(column); // R
    assertThat(tested.getWinner(), is("R"));
}
```

2. 代码

使这个测试通过的代码被加入方法checkWinners中：

```
    if (winner.isEmpty()) {
        String horizontal =
          Stream
          .of(board[row])
          .reduce(String::concat).get();
        if (winPattern.matcher(horizontal)
            .matches())
            winner = colour;
    }
```


5.4.9 需求 8

这是最后一个获胜条件。

> 玩家放入碟片后，如果将其3个以上碟片连成对角线，该玩家将获胜。

1. 测试

我们需要执行有效的走法以满足这个条件。在这里，需要检查棋盘的两条对角线：从右上角到左下角的对角线以及从右下角到左上角的对角线。下面的测试使用了一个列号列表重现受测场景：

```
@Test
public void
when4Diagonal1DiscsAreConnectedThenThatPlayerWins()
{
    int[] gameplay =
       new int[] {1, 2, 2, 3, 4, 3, 3, 4, 4, 5, 4};
    for (int column : gameplay) {
        tested.putDiscInColumn(column);
    }
    assertThat(tested.getWinner(), is("R"));
}

@Test
```

```
public void
when4Diagonal2DiscsAreConnectedThenThatPlayerWins()
{
    int[] gameplay =
       new int[] {3, 4, 2, 3, 2, 2, 1, 1, 1, 1};
    for (int column : gameplay) {
        tested.putDiscInColumn(column);
    }
    assertThat(tested.getWinner(), is("G"));
}
```

2. 代码

同样，需要修改方法checkWinner，以添加新的棋盘检查：

```
    if (winner.isEmpty()) {
        int startOffset = Math.min(column, row);
        int myColumn = column - startOffset,
            myRow = row - startOffset;
        StringJoiner stringJoiner =
        new StringJoiner("");
        do {
            stringJoiner
           .add(board[myRow++][myColumn++]);
        } while (myColumn < COLUMNS &&
                 myRow < ROWS);
        if (winPattern
        .matcher(stringJoiner.toString())
            .matches())
             winner = currentPlayer;
    }

    if (winner.isEmpty()) {
        int startOffset =
        Math.min(column, ROWS - 1 - row);
        int myColumn = column - startOffset,
            myRow = row + startOffset;
        StringJoiner stringJoiner =
        new StringJoiner("");
        do {
            stringJoiner
            .add(board[myRow--][myColumn++]);
        } while (myColumn < COLUMNS &&
                 myRow >= 0);
        if (winPattern
            .matcher(stringJoiner.toString())
            .matches())
            winner = currentPlayer;
    }
```

我们使用TDD创建了一个类，它包含一个构造函数、5个公有方法和6个私有方法。总体而言，这些方法都看起来非常简单且易于理解，但有一个方法较大——检查获胜条件的checkWinner。

这种方法的优点是，提供了很有帮助的测试，可确保未来修改方法时不会改变其行为。代码覆盖率不是最终目标，但我们获得的代码覆盖率确实很高。

另外，为方便测试，我们重构了connect4TDD类的构造函数，使其将一个输出通道作为参数。这样，以后需要修改游戏状态的输出方式时将更容易——无需像传统方法中那样替换所有输出通道。换言之，这个实现的可扩展性更强。

大型项目中，如果发现需要为一个类创建大量测试，应按**单一职责原则**将它分成多个类。鉴于我们将输出工作委托给一个初始化期间以参数方式传入的外部类，因此一种更优雅的解决方案是，创建一个包含高级输出方法的类，从而将输出逻辑和游戏逻辑分开。前面所说的都是使用TDD实现的良好设计带来的好处。

Connect4 > com.packtpublishing.tddjava.ch05connect4 > Connect4TDD

Connect4TDD

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
checkWinner(int, int)		100%		100%	0	13	0	29	0	1
Connect4TDD(PrintStream)		100%		100%	0	2	0	8	0	1
printBoard()		100%		100%	0	2	0	5	0	1
putDiscInColumn(int)		100%		n/a	0	1	0	8	0	1
checkColumn(int)		100%		75%	1	3	0	3	0	1
checkPositionToInsert(int, int)		100%		100%	0	2	0	3	0	1
getCurrentPlayer()		100%		n/a	0	1	0	2	0	1
switchPlayer()		100%		100%	0	2	0	4	0	1
getNumberOfDiscsInColumn(int)		100%		n/a	0	1	0	3	0	1
getNumberOfDiscs()		100%		n/a	0	1	0	2	0	1
isFinished()		100%		100%	0	2	0	1	0	1
getWinner()		100%		n/a	0	1	0	1	0	1
Total	0 of 356	100%	1 of 38	97%	1	31	0	69	0	12

使用这种方法编写的代码可在https://bitbucket.org/vfarcic/tdd-java-ch05-design.git找到。

5.5 小结

本章简要介绍了软件设计和几个基本的设计原则。我们使用两种方法实现了一个功能齐备的桌面游戏Connect4：传统方法和测试驱动开发方法。

我们分析了这两个解决方案的优缺点，并使用了Hamcrest框架强化测试。

最后得出结论：这两种方法都能实现良好设计和良好实践，但TDD可引导开发人员走上更正确的道路。

要更深入地了解本书介绍的主题，强烈推荐你阅读Robert C. Martin的两部著作：《代码整洁之道》和《敏捷软件开发：原则、模式与实践》。

第 6 章

模拟——消除外部依赖

"空谈没有意义，上代码吧。"

——林纳斯·托瓦兹

TDD旨在提高速度。我们要快速验证理念、概念或实现是否有效，还要快速运行所有测试。影响这种速度的主要瓶颈是外部依赖。创建测试所需的DB数据可能需要很长时间；对使用第三方API的代码进行验证的测试执行起来可能很慢；最重要的是，编写满足所有外部依赖的测试可能很复杂，复杂到不值得编写。模拟外部和内部依赖可帮助我们解决这些问题。

本章将以第3章所做的工作为基础，对"井字游戏"进行扩展——使用MongoDB存储数据。但单元测试并不会使用MongoDB，因为所有通信都是模拟的。最后，我们将创建一个集成测试，以验证代码和MongoDB是否很好地集成在一起。

本章涵盖如下主题：

- 模拟；
- Mockito；
- "井字游戏"第二版的需求；
- 开发"井字游戏"第二版；
- 集成测试。

6.1　模拟

无论什么人，只要开发过比Hello World复杂的应用程序，就知道Java代码充斥着依赖。这些依赖可能是其他小组成员编写的类和方法、来自第三方库的类和方法，或要与之通信的外部系统；即便是JDK中的库，也是依赖。我们可能有一个业务层，它与数据访问层通信，而数据访问层又使用数据库驱动程序获取数据。编写单元测试时，依赖的范围更广，甚至要将所有的公有和受保护的方法（即便这些方法位于受测类）视为依赖，进而将其隔离。

在单元测试层面使用TDD时，如果规范需要考虑所有这些依赖，创建将过于复杂，导致测试本身成为瓶颈。开发测试的时间可能急增，导致TDD带来的好处很快被不断增加的成本抵消。更重要的是，这些依赖常常导致测试非常复杂，以至于它们包含的bug比实现本身还多。

单元测试旨在验证单个单元是否正常，而不考虑依赖，TDD中的单元测试尤其如此。对于内部依赖，我们已对其进行过测试，知道它们的行为符合预期；但对于外部依赖，你也必须信任它们——相信它们能够正常工作。就算你不信任，要对其（如JDK包`java.nio`中的类）进行深入测试，工作量也太大。另外，运行功能测试和集成测试时，这些潜在的问题将浮出水面。

为专注于单元，我们必须竭力消除它可能使用的所有依赖，这是通过结合利用设计和模拟实现的。

> **使用模拟对象**
>
> 其优点包括：代码依赖更少；测试执行速度更快。
>
> 要快速执行测试并专注于单个功能单元，必须使用模拟对象。通过模拟受测方法的外部依赖，开发人员能够专注于手头的任务，而无需花时间建立这些依赖。在小组较大或多个小组协同工作的情况下，这些依赖甚至可能还没有开发出来。另外，不使用模拟对象的情况下，测试的执行速度通常很慢。模拟对象非常适合用于代替数据库、其他产品、服务等。

深入介绍模拟对象前，先讲解使用原因。

6.1.1 为何使用模拟对象

下面列出了一些使用模拟对象的原因。

- 对象的结果不确定。例如，每次实例化`java.util.Date`时，得到的结果都不同，我们无法检查其结果是否符合预期：

```
java.util.Date date = new java.util.Date();
date.getTime();// 这个方法返回的结果是什么呢?
```

- 对象不存在。例如，我们可能创建一个接口并对其进行测试，但测试使用这个接口的代码时，实现这个接口的对象可能还没有编写好。
- 对象速度缓慢，需要时间处理。最常见的例子是数据库。我们可能编写了获取所有记录并生成报告的代码，这种操作可能需要几分钟、几小时乃至几天。

上述使用模拟对象的原因适用于所有类型的测试，然而，对于单元测试（尤其是TDD中的单元测试）来说，还有一个原因，可能比其他原因都重要：通过模拟可隔离当前方法使用的所有依赖，这让我们能够专注于单个单元，忽略其调用的代码内部工作原理。

6.1.2　术语

术语可能有点令人迷惑，尤其不同的人会对同样的事物使用不同名称。雪上加霜的是，模拟框架给方法命名时也不统一。

继续介绍前，先简单学习术语。

测试替身是下面各种替身的统称：

- 哑元对象（dummy object）用于替换真正的方法参数；
- 测试存根（test stub）用于将实际对象替换为测试特定对象，以便向受测系统提供所需的间接输入；
- 测试间谍（test spy）记录**受测系统**（**SUT**）向另一个组件发出的间接输出调用，让测试随后能够进行验证；
- 模拟对象（mock object）用于将受测系统依赖的对象替换为测试特定对象，以验证SUT是否正确使用；
- 伪造对象（fake object）用于将受测系统依赖的组件替换为量级更轻的实现。

如果你仍感到迷惑，没关系，其实这样的人不止你一个。雪上加霜的是，不同的框架和作者并未就这些定义和命名标准达成一致。这导致术语混乱而不一致，前述术语的定义并非被大家普遍接受。

出于简化考虑，本书将始终使用Mockito（我们使用的框架）采用的命名约定。这样，你使用的方法将与后面学到的术语保持一致。我们将继续使用统称“模拟”，而其他人可能称之为“测试替身”。另外，我们将使用模拟对象或间谍表示`Mockito`方法。

6.1.3　模拟对象

模拟对象模拟实际对象（通常很复杂）的行为，让我们能够创建对象以替换实现代码中使用的实际对象。模拟对象期望使用指定参数调用指定方法，以返回预期的结果，它预先知道应发生什么情况以及我们期望它做何反应。

下面看一个简单的示例：

```
TicTacToeCollection collection =
    mock(TicTacToeCollection.class);

assertThat(collection.drop()).isFalse();

doReturn(true).when(collection).drop();

assertThat(collection.drop()).isTrue();
```

首先，我们将`collection`定义为一个可替换`TicTacToeCollection`的模拟对象。此时这

个模拟对象的所有方法都是伪造的（fake），在Mockito中意味着返回默认值。第二行也确认了这一点，它断言方法`drop`返回`false`。接下来，我们指定模拟对象collection应在其方法`drop`被调用时返回`true`。最后，我们断言方法`drop`返回`true`。

这个示例中，我们创建一个返回默认值的模拟对象，然后指定它的一个方法应返回的值。自始至终都未使用实际对象。

本章后面将使用间谍，其逻辑与模拟对象相反：除非另有说明，否则使用实际方法。稍后开始扩展“井字游戏”应用程序时，你将更深入地了解模拟。现在先来看看Java模拟框架Mockito。

6.2　Mockito

Mockito是一个模拟框架，其API简单而整洁。使用Mockito生成的测试直观而易于理解，它提供了三个主要的静态方法：

- `mock()`：用于创建模拟对象，还可使用`when()`和`given()`指定这些模拟对象的行为。
- `spy()`：可用于实现部分模拟。除非另有说明，否则间谍对象调用实际方法。与模拟对象一样，对于间谍对象的每个公有或受保护的方法（静态方法除外），都可设置其行为。主要差别在于，`mock()`创建一个完全伪造的对象，而`spy()`使用实际对象。
- `verify()`：用于检查调用方法时提供的是否是指定参数，这是一种断言。

后面编写“井字游戏”第二版时，将更深入地介绍Mockito，但在此之前，先简单介绍这个游戏新增的需求。

6.3　“井字游戏”第二版的需求

“井字游戏”第二版的需求很简单：添加永久性存储，让玩家能够保存游戏的当前状态，以便以后接着玩。我们将使用MongoDB实现这个目标。

> 在“井字游戏”中添加MongoDB永久性存储。

6.4　开发“井字游戏”第二版

我们将在第3章开发的“井字游戏”基础上继续，这个游戏当前的完整源代码可在https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo.git找到。请在**IntelliJ IDEA**中使用**VCS > Checkout from Version Control > Git**复制这些代码。与其他项目一样，我们首先要在`build.gradle`中添加依赖：

```
dependencies {
    compile 'org.jongo:jongo:1.1'
    compile 'org.mongodb:mongo-java-driver:2.+'
    testCompile 'junit:junit:4.11'
    testCompile 'org.mockito:mockito-all:1.+'
}
```

导入MongoDB驱动程序的代码应该是不言自明的。Jongo是一个很有用的实用方法集，让你能够像使用Mongo查询语言一样使用Java代码访问MongoDB数据库。对于测试，我们将继续使用JUnit，同时使用Mockito模拟对象、间谍和验证。

你将发现，我们要等到开发接近尾声才安装MongoDB。有了Mockito，我们不再需要MongoDB，因为我们将模拟所有Mongo依赖。

指定依赖后，别忘了在IDEA **Gradle projects**对话框中刷新。

源代码可在Git仓库`tdd-java-ch06-tic-tac-toe-mongo`的分支`00-prerequisites`（https://bitbucket.zorg/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/00-prerequisites）中找到。

做好准备工作后，下面着手处理第一个需求。

6.4.1　需求 1

我们需要将每步棋都存储到数据库。鉴于所有游戏逻辑都已实现，这项工作应该很简单。尽管如此，这个示例将淋漓尽致地展示模拟对象的用法。

> 实现一个保存单步棋——包含轮次、X和Y坐标以及玩家（X或O）——的选项。

1. 规范和实现

我们应首先定义用于表示数据存储模式（schema）的Java bean。这没什么特殊的，故省略该部分，只对此稍加说明。

不要花太多时间定义针对Java样板式代码的规范。我们的bean实现包含重写的`equals`和`hashCode`，它们都是IDEA自动生成的。除了满足对两个相同类型的对象进行比较的需求外，没有其他实际价值（后面的规范中将使用这种比较）。TDD可帮助我们改善设计质量、编写更好的代码，但编写15~20个规范以定义原本可由IDE生成的样板式代码（如方法equals），并不能帮助我们实现这样的目标。要掌握TDD，不仅要学会如何编写规范，还要知道什么样的规范不值得编写。

话虽如此，要了解完整的bean规范和实现，请参阅源代码。

这些源代码可在Git仓库tdd-java-ch06-tic-tac-toe-mongo的分支01-bean（https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/01-bean）中找到。包含bean规范和实现的类分别为TicTacToeBeanSpec和TicTacToeBean。

下面进入更有趣的部分，编写将数据存储到MongoDB的相关规范（这部分也没有使用模拟对象、间谍和验证）。

对于这个需求，我们将在com.packtpublishing.tddjava.ch03tictactoe.mongo包中创建两个新类：TicTacToeCollectionSpec（在src/test/java中）和TicTacToeCollection（在src/main/java中）。

2. 规范

我们应规范将使用的数据库名称：

```
@Test
public void
whenInstantiatedThenMongoHasDbNameTicTacToe() {
    TicTacToeCollection collection =
            new TicTacToeCollection();
    assertEquals(
            "tic-tac-toe",
            collection.getMongoCollection()
             .getDBCollection().getDB().getName());
}
```

实例化一个TicTacToeCollection类，并验证数据库名称是我们期望的。

3. 实现

实现非常简单，如下所示：

```
private MongoCollection mongoCollection;
protected MongoCollection getMongoCollection() {
    return mongoCollection;
}
public TicTacToeCollection()
        throws UnknownHostException {
    DB db = new MongoClient().getDB("tic-tac-toe");
    mongoCollection =
            new Jongo(db).getCollection("bla");
}
```

实例化TicTacToeCollection类时，使用指定数据库名称（tic-tac-toe）创建一个新的MongoCollection对象，并将其赋给一个局部变量。

请耐心等待，再完成一个规范，我们就将进入有趣的部分，届时将使用模拟对象和间谍。

4. 规范

前一个实现使用了集合名bla，因为Jongo要求我们必须指定一个字符串。下面创建一个规范，以指定将使用的Mongo集合的名称：

```
@Test
public void
whenInstantiatedThenMongoCollectionHasNameGame() {
    TicTacToeCollection collection = new
            TicTacToeCollection();
    assertEquals(
            "game",
            collection.getMongoCollection()
                    .getName());
}
```

这个规范与前一个几乎完全相同，且很可能是不言自明的。

5. 实现

为实现这个规范，只需修改用于设置集合名的字符串：

```
public TicTacToeCollection()
        throws UnknownHostException {
    DB db = new MongoClient().getDB("tic-tac-toe");
    mongoCollection =
            new Jongo(db).getCollection("game");
}
```

6. 重构

你可能以为重构是专为实现代码准备的，但由其目标（更易读、更佳、速度更快的代码）可知，重构既可用于实现，也可用于规范。

前面两个规范都实例化了TicTacToeCollection类，我们可以将这些重复的代码移到一个用@Before注解的方法中。最终的效果没变（运行每个用@Test注解的方法前，都将实例化TicTacToeCollection类），但消除了重复代码。由于后面的规范也需要这样的实例化，因此消除重复将在以后带来更多好处。另外，我们还需避免反复编写引发UnknownHostException异常的代码：

```
TicTacToeCollection collection;

@Before
public void before() throws UnknownHostException {
    collection = new TicTacToeCollection();
}
@Test
public void
whenInstantiatedThenMongoHasDbNameTicTacToe() {
```

```
//        throws UnknownHostException {
//    TicTacToeCollection collection = new
//    TicTacToeCollection();
        assertEquals("tic-tac-toe",
            collection.getMongoCollection()
                .getDBCollection().getDB()
                    .getName());
}

@Test
public void whenInstantiatedThenMongoHasNameGame()
{
//        throws UnknownHostException {
//    TicTacToeCollection collection = new
//    TicTacToeCollection();
    assertEquals("game",
            collection.getMongoCollection()
                .getName());
}
```

使用设置和拆除方法

这样做的好处是，这些方法让我们能够分别在类或各测试方法之前和之后执行准备（设置）和销毁（拆除）代码。

我们常常需要在测试类或其每个方法之前执行一些代码，为此，JUnit提供了注解`@BeforeClass`和`@Before`，其中注解`@BeforeClass`在类被加载前（第一个测试方法运行前）执行与它相关联的方法，而`@Before`在每个测试运行前执行与它相关联的方法。这两个注解用于测试存在前置条件的情形，最常见的例子是在数据库（很可能位于内存）中设置测试数据。与这两个注解对应的是`@After`和`@AfterClass`，它们的主要用途是销毁设置阶段或其他测试创建的数据或状态。每个测试都应独立于其他测试，另外，任何测试都不应受其他测试的影响。拆除阶段有助于确保系统就像未执行任何测试一样。

下面做一些模拟、监视（spying）和验证工作！

7. 规范

我们需要创建一个方法，将数据保存到MongoDB。研究Jongo文档可发现，方法`MongoCollection.save`所做的就是这样的工作。它将任何对象作为参数，并使用Jackson将其转换为MongoDB使用的JSON。这里的重点是，经过对Jongo的一番处理，我们决定使用（更重要的是信任）这个库。

可以通过两种方式编写Mongo规范，其中一种较为传统，适合**端到端**（**E2E**）测试和集成测试：启动一个MongoDB实例，调用Jongo方法save，查询数据库并确认数据确实保存到数据库。这还没有完，因为每个测试前都需要清理数据库，确保它处于未被之前的测试污染的初始状态。

最后，所有测试都执行完毕后，还可能要停止MongoDB实例，以释放服务器资源，让其他任务可用。

你可能猜到了，以这种方式编写的测试需要做很多工作。不仅如此，测试的执行时间也将激增。运行一个与数据库通信的测试不需要很长时间，运行10个这样的测试通常也很快，但运行数百乃至数千个这样的测试将需要很长时间。如果运行所有单元测试需要很长时间，结果将如何呢？大家将失去耐心，开始将它们分组甚至完全放弃TDD。将测试分组意味着我们无法确信没有破坏任何东西，因为始终只测试了部分代码。至于放弃TDD，那可不是我们要力图实现的目标。然而，如果运行测试需要很长时间，那就完全有理由认为开发人员不愿等到它们运行完毕后才进入下一个规范。而一旦开发人员这样做，践行的就不再是TDD。单元测试的运行时间多长算合理呢？没有放之四海皆准的规则，但一条经验是：如果时间超过10~15秒，就应对此感到担忧，并花时间对测试进行优化。

测试应能快速运行

这样做的好处是，测试将被频繁运行。

如果测试的运行时间很长，开发人员将不再运行它们，或者只运行与所做修改相关的很少一部分测试。除促使开发人员使用它们外，测试运行速度快的另一个好处是可快速提供反馈。问题发现得越早，修复就越容易，因为此时对导致问题的代码还记忆犹新。如果等待测试执行完毕期间，开发人员已开始着手处理下一个功能，他可能决定将修复问题的工作推迟到完成新功能后去做。另一方面，如果他放下手头的工作去修复问题，将因调整思路而浪费时间。

既然使用真实的数据库运行单元测试不是好的选择，那么有什么替代办法吗？模拟和监视！这个示例中，我们知道应调用第三方库的哪个方法，还投入足够的时间确认这个库是可信任的（后者还将执行集成测试确认其是可信任的）。知道如何使用这个库后，就可将工作范围限定为验证是否正确调用了这个库。

下面就来试一试。

首先，应该修改既有代码，将`TicTacToeCollection`对象替换为间谍：

```
import static org.mockito.Mockito.*;
...
@Before
public void before() throws UnknownHostException {
    collection = spy(new TicTacToeCollection());
}
```

对类进行监视称为“**部分模拟**”。被监视时，类的行为与正常实例化时完全相同，主要差别在于，可以应用部分模拟，对一个或多个方法进行替换。大多数情况下，我们都对要测试的类进

行监视，因为想保留其所有功能；同时提供这样的选择，即能够在需要时模拟其一部分。

下面编写规范本身，它可能如下所示：

```
@Test
public void
whenSaveMoveThenInvokeMongoCollectionSave() {
    TicTacToeBean bean =
            new TicTacToeBean(3, 2, 1, 'Y');
    MongoCollection mongoCollection =
            mock(MongoCollection.class);
    doReturn(mongoCollection).when(collection)
            .getMongoCollection();
    collection.saveMove(bean);
    verify(mongoCollection, times(1)).save(bean);
}
```

mock、doReturn和verify等静态方法都来自org.mockito.Mockito类。

首先，创建一个新的TicTacToeBean对象，这没有什么特别之处。接下来，根据MongoCollection创建一个模拟对象。鉴于我们已经确定，进行单元测试时应避免直接与数据库通信，而通过模拟这个依赖可达成这个目标。模拟将真实类转换为模拟类，在使用mongoCollection的类看来，它就像一个真实的类。但在幕后，它的所有方法都是shallow，即什么都不做：就像重写了这个类，将其所有方法都改为空：

```
MongoCollection mongoCollection =
      mock(MongoCollection.class);
```


接下来指出，每当调用间谍对象collection的方法getMongoCollection时，都应返回模拟对象mongoCollection。换言之，我们让类使用伪造的集合而不是真实集合：

```
doReturn(mongoCollection).when(collection)
.getMongoCollection();
```

接下来，调用要测试的方法：

```
collection.saveMove(bean);
```

最后，需要验证是否正确调用了Jongo库，且只调用了1次：

```
verify(mongoCollection, times(1)).save(bean);
```

下面尝试实现这个规范。

8. 实现

为更好地理解刚才编写的规范，只提供部分实现——创建空方法saveMove。在不实现这个规范的情况下，这能够使代码通过编译：

```
public void saveMove(TicTacToeBean bean) {
}
```

你运行所有规范（执行命令`gradle test`）时，结果如下：

```
Wanted but not invoked:
mongoCollection.save(
    Turn: 3; X: 2; Y: 1; Player: Y
);
```

Mockito指出，从规范可知，我们期望方法`mongoCollection.save`被调用，但这个期望未得到满足。由于测试失败，我们需要回去完成实现。使用TDD时，最大的罪过之一是，在测试未通过的状态下就去做其他事情。

仅当所有测试都通过才编写新测试

这样做的好处是：专注于小型工作单元，而实现代码几乎始终处于能够运行的状态。

有时候开发人员可能很想编写多个测试再实现，还有些时候开发人员会忽略测试检测到的问题，接着添加新功能。这些做法都必须尽可能避免。大多数情况下，违反这个原则都将引入需要"连本带息"一起偿还的技术债务。确保实现代码几乎始终像预期的那样工作是TDD的目标之一。为按期交付或避免超预算，有些项目违背这个原则，将时间都用于开发新功能；而对于测试失败的代码，会推迟其修复工作。这些项目最终都难逃推迟交付的命运。

下面修改前面的实现，如下所示：

```
public void saveMove(TicTacToeBean bean) {
    getMongoCollection().save(null);
}
```

如果再次运行规范，结果将如下所示：

```
Argument(s) are different! Wanted:
mongoCollection.save(
    Turn: 3; X: 2; Y: 1; Player: Y
);
```

这次调用了期望的方法，但传递给它的参数不符合预期。在规范中，我们将期望设置为一个bean（`new TicTacToeBean(3, 2, 1, 'Y')`）；而在实现中，我们传递的是null。Mockito验证不仅能够指出是否调用了正确的方法，还能指出传递给这个方法的参数是否正确。

这个规范的正确实现如下所示：

```
public void saveMove(TicTacToeBean bean) {
    getMongoCollection().save(bean);
}
```

这次所有规范都通过了，可以接着编写下一个规范。

9. 规范

将方法saveMove的返回类型改为布尔值：

```
@Test
public void whenSaveMoveThenReturnTrue() {
    TicTacToeBean bean =
            new TicTacToeBean(3, 2, 1, 'Y');
    MongoCollection mongoCollection =
            mock(MongoCollection.class);
    doReturn(mongoCollection).when(collection)
            .getMongoCollection();
    assertTrue(collection.saveMove(bean));
}
```

10. 实现

这个规范的实现非常简单，只需修改指定方法的返回类型即可。别忘了，“使用尽可能简单的解决方案”是TDD规则之一。最简单的解决方案是返回true，如下所示：

```
public boolean saveMove(TicTacToeBean bean) {
    getMongoCollection().save(bean);
    return true;
}
```

11. 重构

你可能注意到，前面两个规范的开头两行代码是重复的。我们可以重构这些规范，将这些相同代码移到@Before注解的方法中：

```
TicTacToeCollection collection;
TicTacToeBean bean;
MongoCollection mongoCollection;

@Before
public void before() throws UnknownHostException {
    collection = spy(new TicTacToeCollection());
    bean = new TicTacToeBean(3, 2, 1, 'Y');
    mongoCollection = mock(MongoCollection.class);
}
...
@Test
public void
whenSaveMoveThenInvokeMongoCollectionSave() {
//    TicTacToeBean bean =
//    new TicTacToeBean(3, 2, 1, 'Y');
//    MongoCollection mongoCollection =
//        mock(MongoCollection.class);
    doReturn(mongoCollection).when(collection)
```

```
            .getMongoCollection();
    collection.saveMove(bean);
    verify(mongoCollection, times(1)).save(bean);
}

@Test
public void whenSaveMoveThenReturnTrue() {
//    TicTacToeBean bean =
//    new TicTacToeBean(3, 2, 1, 'Y');
//    MongoCollection mongoCollection =
//        mock(MongoCollection.class);
    doReturn(mongoCollection).when(collection)
            .getMongoCollection();
    assertTrue(collection.saveMove(bean));
}
```

12. 规范

现在处理使用MongoDB时可能出错的问题。例如，出现异常时，我们可能要从方法saveMove返回false：

```
@Test
public void
givenExceptionWhenSaveMoveThenReturnFalse() {
    doThrow(new MongoException("Bla"))
            .when(mongoCollection)
            .save(any(TicTacToeBean.class));
    doReturn(mongoCollection).when(collection)
      .getMongoCollection();
    assertFalse(collection.saveMove(bean));
}
```

这里引入另一个Mockito方法——doThrow，它类似于doReturn，但在when设置的条件满足是"引发异常"。这个规范在mongoCollection类的方法save被调用时引发MongoException异常，这让我们能够断言方法saveMove在出现异常时返回false。

13. 实现

实现很简单，只需添加一个try/catch块即可：

```
public boolean saveMove(TicTacToeBean bean) {
    try {
        getMongoCollection().save(bean);
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

14. 规范

这个应用程序非常简单，只能保存一盘棋——至少目前是这样的。因此，每次创建新实例时，都需重新开始，将数据库中存储的所有数据都删除。为此，最简单的方式是删除整个MongoDB集合。Jongo提供的方法`MongoCollection.drop()`可用于完成这种任务。我们将创建一个类似于`saveMove`的新方法——`drop()`。

如果以前没有使用过Mockito、MongoDB和Jongo，则很可能无法自行完成本章练习，而只能按书中说的做。若果真如此，现在你可能想改弦易辙，尝试自己编写规范和实现。

我们需要验证在`TicTacToeCollection`类的方法`drop()`中调用了`MongoCollection.drop()`。请先尝试自己编写这个规范，再查看下面的代码。这个规范应该与`save`方法相关的规范几乎相同：

```
@Test
public void
whenDropThenInvokeMongoCollectionDrop() {
    doReturn(mongoCollection).when(collection)
            .getMongoCollection();
    collection.drop();
    verify(mongoCollection).drop();
}
```

15. 实现

这是一个包装器方法，因此这个规范实现起来应该非常容易：

```
public void drop() {
    getMongoCollection().drop();
}
```

16. 规范

再实现两个规范，`TicTacToeCollection`类就完成。

下面确保我们在正常情况下返回`true`：

```
@Test
public void whenDropThenReturnTrue() {
    doReturn(mongoCollection).when(collection)
            .getMongoCollection();
    assertTrue(collection.drop());
}
```

17. 实现

如果说使用TDD时，规范实现起来都很简单，那是因为我们有意为之。将任务分成很小的实体，使得大多数规范的实现都是小菜一碟。这个规范的实现也不例外：

```
public boolean drop() {
    getMongoCollection().drop();
    return true;
}
```

18. 规范

最后，确保方法drop在发生异常时返回false：

```
@Test
public void
givenExceptionWhenDropThenReturnFalse() {
    doThrow(new MongoException("Bla"))
            .when(mongoCollection)
            .drop();
    doReturn(mongoCollection).when(collection)
            .getMongoCollection();
    assertFalse(collection.drop());
}
```

19. 实现

为实现这个规范，只需添加一个try/catch块：

```
public boolean drop() {
    try {
        getMongoCollection().drop();
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

有了这个实现后，TicTacToeCollection类就编写好了，它是我们的主类和MongoDB的中间层。

源代码可在Git仓库tdd-java-ch06-tic-tac-toe-mongo的分支02-save-move（https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/02-save-move）中找到。规范和实现类分别为TicTacToeCollectionSpec和TicTacToeCollection。

6.4.2　需求 2

下面在主类TicTacToe中使用TicTacToeCollection方法。每当玩家成功落子后，我们都需要将这步棋保存到数据库。另外，每当实例化TicTacToe类时，都需将集合删除，以免旧数据影响新游戏。我们原本可以做得更精致些，但本章目的是学习如何使用模拟，因此这里只考虑这些需求。

将每步棋都保存到数据库，并确保开始新游戏时删除旧数据。

下面先做些准备工作。

1. 规范

与MongoDB通信的方法都位于TicTacToeCollection类，因此我们需要确保对其进行了实例化。这个规范可以这样编写：

```
@Test
public void whenInstantiatedThenSetCollection() {
    assertNotNull(
            ticTacToe.getTicTacToeCollection());
}
```

实例化TicTacToe的工作已经在用@Before注解的方法中完成。这个规范中，我们确保也实例化了TicTacToeCollection。

2. 实现

这个实现没什么特别之处，只需修改默认构造函数，将一个新的TicTacToeCollection实例赋给变量ticTacToeCollection。

首先，添加一个类型为TicTacToeCollection的局部变量及其获取函数：


```
private TicTacToeCollection ticTacToeCollection;

protected TicTacToeCollection
        getTicTacToeCollection() {
    return ticTacToeCollection;
}
```

现在，余下的全部工作就是在主类被实例化时，实例化一个新集合，并将其赋给这个变量：

```
public TicTacToe() throws UnknownHostException {
    this(new TicTacToeCollection());
}
protected TicTacToe
        (TicTacToeCollection collection) {
    ticTacToeCollection = collection;
}
```

我们还提供了另一种实例化这个类的方式：通过参数传入一个TicTacToeCollection对象。这将在规范内部派上用场——使用它轻松传入模拟的集合。

现在回到规范类，在其中使用这个新的构造函数。

3. 重构规范

如下所示使用新的TicTacToe构造函数：

```
private TicTacToeCollection collection;

@Before
public final void before()
        throws UnknownHostException {
    collection = mock(TicTacToeCollection.class);
//    ticTacToe = new TicTacToe();
    ticTacToe = new TicTacToe(collection);
}
```

现在，所有规范都将使用TicTacToeCollection的模拟版。还有其他注入模拟依赖的方法，如使用Spring，但我们认为简单的方式胜过使用复杂的框架。

4. 规范

每当玩家落子后，都需将这步棋保存到数据库。可以这样编写相应规范：

```
@Test
public void whenPlayThenSaveMoveIsInvoked() {
    TicTacToeBean move =
            new TicTacToeBean(1, 1, 3, 'X');
    ticTacToe.play(move.getX(), move.getY());
    verify(collection).saveMove(move);
}
```

至此，你应该熟悉了Mockito，但下面还是详细介绍一下这些代码，以帮助你复习：

(1) 首先，实例化一个TicTacToeBean对象，因为它包含集合期望的数据：

```
TicTacToeBean move = new TicTacToeBean(1, 1, 3, 'X');
```

(2) 接下来，该实际落子了：

```
ticTacToe.play(move.getX(), move.getY());
```

(3) 最后，需要验证确实调用了方法saveMove：

```
verify(collection, times(1)).saveMove(move);
```

与本章一直做的一样，我们隔离所有外部调用，只专注于当前要测试的单元（play）。别忘了，这种隔离仅限于公有和受保护的方法。实际实现中，我们可能在公有方法play中调用saveMove，也可能在前面重构时编写的一个私有方法中调用它。

5. 实现

这个规范带来两个挑战。首先，我们该在哪里调用方法saveMove呢？在私有方法setBox

中调用看起来是个不错的选择。这个方法中，我们检查落子位置是否有效，如果有效，就可调用方法saveMove。然而，方法saveMove将一个bean作为参数，而setBox当前接受的参数为x、y和lastPlayer，因此我们需要修改方法setBox的签名。

方法setBox当前是这样的：

```
private void setBox(int x, int y, char lastPlayer)
{
    if (board[x - 1][y - 1] != '\0') {
        throw new RuntimeException(
                "Box is occupied");
    } else {
        board[x - 1][y - 1] = lastPlayer;
    }
}
```

我们需要将其修改如下：

```
private void setBox(TicTacToeBean bean) {
    if (board[bean.getX() - 1][bean.getY() - 1]
            != '\0') {
        throw new RuntimeException(
                "Box is occupied");
    } else {
        board[bean.getX() - 1][bean.getY() - 1] =
                lastPlayer;
        getTicTacToeCollection().saveMove(bean);
    }
}
```


修改setBox的签名后，还有其他几个地方也需要修改。由于方法play调用了setBox，我们需要在其中实例化bean：

```
public String play(int x, int y) {
    checkAxis(x);
    checkAxis(y);
    lastPlayer = nextPlayer();
//    setBox(x, y, lastPlayer);
    setBox(new TicTacToeBean(1, x, y, lastPlayer));
    if (isWin(x, y)) {
        return lastPlayer + " is the winner";
    } else if (isDraw()) {
        return RESULT_DRAW;
    } else {
        return NO_WINNER;
    }
}
```

你可能注意到了，我们将轮次设置成常量1。由于还没有规范指定要以其他方式设置轮次，因此这里采取了最简单的方式。轮次的问题将在后面处理。

所有这些修改都非常简单，实现起来所需的时间很短。如果修改的范围更大，我们可能采取不同的做法：先做简单的修改，再通过重构实现最终解决方案。别忘了速度是关键，我们不想长时间纠缠于不影响测试通过的实现。

6. 规范

如果无法保存将下的棋，该如何办呢？辅助方法saveMove根据MongoDB操作的结果返回true或false，而我们可能想在它返回false时引发异常。

先做重要的事情。我们应修改方法before的实现，确保saveMove默认返回true：

```
@Before
public final void before()
        throws UnknownHostException {
    collection = mock(TicTacToeCollection.class);
    doReturn(true)
            .when(collection)
            .saveMove(any(TicTacToeBean.class));
    ticTacToe = new TicTacToe(collection);
}
```

给模拟的集合指定默认行为（在saveMove被调用时返回true）后，便可以接着编写这个规范了：

```
@Test
public void
whenPlayAndSaveReturnsFalseThenThrowException() {
    doReturn(false).when(collection).
            saveMove(any(TicTacToeBean.class));
    TicTacToeBean move =
            new TicTacToeBean(1, 1, 3, 'X');
    exception.expect(RuntimeException.class);
    ticTacToe.play(move.getX(), move.getY());
}
```

我们使用Mockito指定，saveMove被调用时返回false。此处，我们不关心saveMove具体是怎么调用的，因此向它传递参数any(TicTacToeBean.class)。any()也是一个静态的Mockito方法。

万事俱备后，像第3章那样使用一个JUnit异常。

7. 实现

下面做简单的检查，在结果不符合预期时引发RuntimeException：

```
private void setBox(TicTacToeBean bean) {
    if (board[bean.getX() - 1][bean.getY() - 1]
            != '\0') {
        throw new RuntimeException(
```

```
                "Box is occupied");
    } else {
        board[bean.getX() - 1][bean.getY() - 1] =
                lastPlayer;
//        getTicTacToeCollection().saveMove(bean);
        if (!getTicTacToeCollection()
                .saveMove(bean)) {
            throw new RuntimeException(
                    "Saving to DB failed");
        }
    }
}
```

8. 规范

你可能还记得，前面总是以硬编码的方式将轮次设置为1。下面修复这个问题。

我们可以调用方法play两次，并验证轮次从1变成2：

```
@Test
public void
whenPlayInvokedMultipleTimesThenTurnIncreases() {
    TicTacToeBean move1 =
            new TicTacToeBean(1, 1, 1, 'X');
    ticTacToe.play(move1.getX(), move1.getY());
    verify(collection, times(1)).saveMove(move1);
    TicTacToeBean move2 =
            new TicTacToeBean(2, 1, 2, 'O');
    ticTacToe.play(move2.getX(), move2.getY());
    verify(collection, times(1)).saveMove(move2);
}
```

9. 实现

与以TDD方式做的其他任何事情一样，实现也非常容易：

```
private int turn = 0;
...
public String play(int x, int y) {
    checkAxis(x);
    checkAxis(y);
    lastPlayer = nextPlayer();
    setBox(new TicTacToeBean(++turn, x, y,
            lastPlayer));
    if (isWin(x, y)) {
        return lastPlayer + " is the winner";
    } else if (isDraw()) {
        return RESULT_DRAW;
    } else {
        return NO_WINNER;
    }
}
```

10. 练习

还有几个规范及其实现没有完成。每当实例化TicTacToe类时，都应调用方法drop()；我们还应确保，drop()返回false时引发RuntimeException异常。这些规范及其实现就作为练习留给你去完成吧。

源代码可在Git仓库tdd-java-ch06-tic-tac-toe-mongo的分支03-mongo（https://bitbucket.org/vfarcic/tdd-java-ch06-tic-tac-toe-mongo/branch/03-mongo）中找到。规范和实现类分别为TicTacToeSpec和TicTacToe。

6.5　集成测试

前面编写了大量单元测试，并对大量依赖采取了信任的态度。我们逐个规范并实现单元；编写规范时，我们将除当前单元外的东西都隔离开来，并验证单元正确调用了其他单元。然而，现在该验证所有这些单元都能与MongoDB通信了。我们可能犯了错，更重要的是，可能还没有让MongoDB运行起来。如果部署应用程序后发现没有运行这个数据库，或者没有正确设置配置（IP、端口等），后果将是灾难性的。

你可能猜到了，集成测试旨在验证各个组件、应用程序、系统等的集成情况。你可能还记得本书前面说的测试金字塔，它指出单元测试最容易编写，运行速度也最快。因此，我们应仅将其他类型的测试用于UT未覆盖的地方。

我们应将集成测试分离出来，以便能够偶尔运行它们（将代码加入仓库前或作为持续集成的一部分），同时使用单元测试提供持续反馈。

6.5.1　分离测试

如果遵循某种约定，就很容易在Gradle中将测试分离开来。例如，我们可以将测试放在不同的目录和包中，或者使用不同的文件名后缀。此处选择后一种方法：对于所有规范类，给它们命名时都加上后缀Spec（如TicTacToeSpec）。我们可以制定一条规则，规定所有集成测试都在名称中使用后缀Integ。

明白这一点后，下面修改文件build.gradle。

首先，我们告诉Gradle，test任务应只使用名称以Spec结尾的类：

```
test {

    include '**/*Spec.class'
}
```

接下来，创建一个新任务——testInteg：

```
task testInteg(type: Test) {
    include '**/*Integ.class'
}
```

在文件build.gradle中添加这两项内容后，我们依然拥有测试任务，这些任务贯穿本书，但现在仅限于规范（单元测试）。另外，要想运行所有集成测试，可在IDEA窗口Gradle projects中单击任务testInteg，也可从命令提示符执行如下命令：

```
gradle testInteg
```

下面编写一个简单的集成测试。

6.5.2　集成测试

包com.packtpublishing.tddjava.ch03tictactoe的目录src/test/java下，创建一个名为TicTacToeInteg的类。我们知道，不能连接到数据库时，Jongo将引发异常。因此这个测试类可以很简单，如下所示：

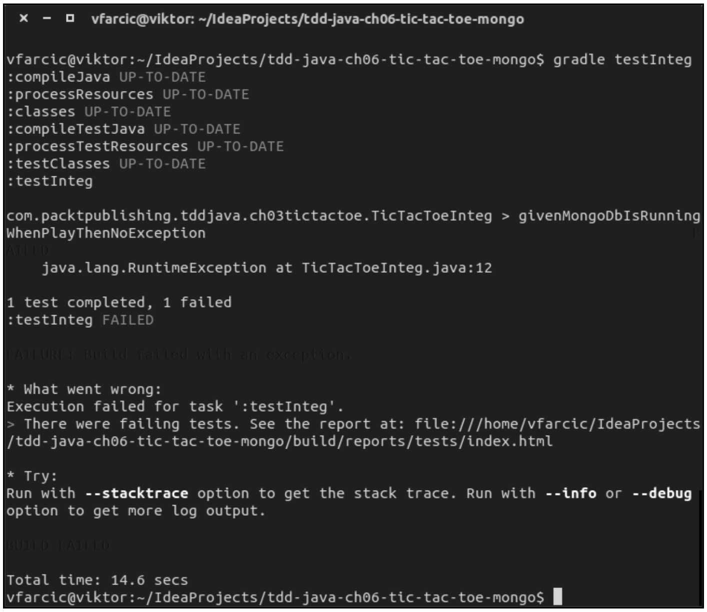
```
import org.junit.Test;
import java.net.UnknownHostException;
import static org.junit.Assert.*;

public class TicTacToeInteg {

    @Test
    public void
        givenMongoDbIsRunningWhenPlayThenNoException()
            throws UnknownHostException {
        TicTacToe ticTacToe = new TicTacToe();
        assertEquals(TicTacToe.NO_WINNER,
        ticTacToe.play(1, 1));
    }

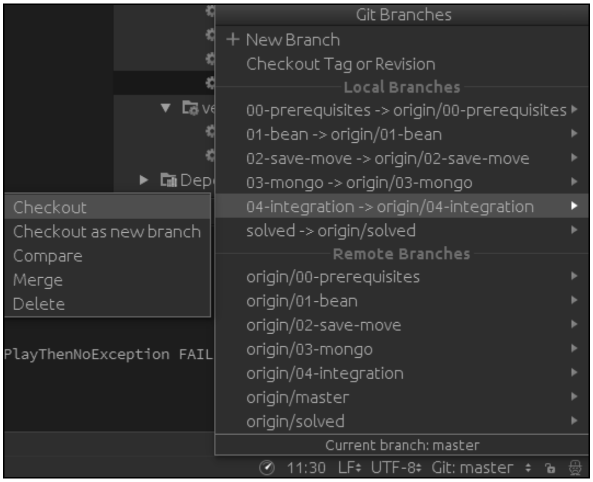
}
```

调用assertEquals只是一种预防措施，这个测试的真正目标是确保不会引发异常。由于我们没有启动MongoDB（除非你未雨绸缪，自己这样做了；如果是这样，你应将其停止），测试将失败。



知道这个集成测试管用（换言之，MongoDB没有启动时它确实失败了）后，我们在启动MongoDB的情况下再次运行它。为启动MongoDB，使用Vagrant创建一个使用操作系统Ubuntu的虚拟机，并将MongoDB作为docker运行。

请确保签出分支`04-integration`。



从命令提示符运行如下命令：

```
$ vagrant up
```

请耐心等待VM启动完毕（首次执行这个命令时可能需要一段时间，连接带宽较低时尤其如此），再运行集成测试。

```
vfarcic@viktor: ~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo
vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$ gradle testInteg
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:testInteg

BUILD SUCCESSFUL

Total time: 4.46 secs
vfarcic@viktor:~/IdeaProjects/tdd-java-ch06-tic-tac-toe-mongo$
```

测试成功，现在我们深信，确实成功集成了MongoDB。

这个集成测试非常简单，实际工作中需要做的集成测试更多。例如，我们可能查询数据库，确认正确存储了数据。然而，本章的目的是让你学会如何模拟，并知道不能完全依赖单元测试。下一章将更深入地探讨集成测试和功能测试。

可在Git仓库`tdd-java-ch06-tic-tac-toe-mongo`的分支`04-integration`（https://bitbucket.org/vfarcic/ tdd-java-ch06-tic-tac-toe-mongo/branch/04-integration）中找到源代码。

6.6 小结

模拟和监视技术用于隔离其他代码和第三方库，需要快速编写代码和运行测试时必不可少。如果不使用模拟对象，测试通常将复杂得难以编写，且运行速度很慢，导致TDD几乎无法进行。测试缓慢意味着无法在每次编写新规范后都运行所有测试，这将导致我们对测试的信任度下降，因为只运行了部分测试。

模拟不仅是一种很有用的外部依赖隔离方式，还可将你自己编写的代码与当前单元隔离。

本章介绍了Mockito，在我们看来，这个框架在平衡功能性和易用性方面是最好的。建议你深入研究这个框架的文档（http://mockito.org/）和其他几个最流行的Java模拟框架——EasyMock（http://easymock.org/）、JMock（http://www.jmock.org/）和PowerMock（https://code.google.com/p/powermock/）。

第 7 章

BDD——与整个团队协作

“我并非伟大的程序员，而只是有良好习惯的优秀程序员。”

——肯特·贝克

前面介绍的都是只适用于开发人员的技巧，而未涉及客户、业务代表和其他无法理解代码的相关方。

TDD涉及的范围比我们前面所做的大得多。我们可定义需求，与客户讨论需求并就开发内容达成一致；我们可以将这些需求变成可执行的，以驱动和验证开发工作；我们可使用普通语言编写验收测试。所有这些以及其他任务都可使用名为行为驱动开发（BDD）的TDD变种完成。

我们将使用BDD方法开发一个书店应用程序：使用自然语言定义验收测试，分别创建每项功能的实现，通过运行BDD场景确认它们能够正常工作，并在必要时重构代码以达到要求的质量等级。这个过程中，我们依然采用“红灯–绿灯–重构”流程，这是TDD的精髓所在。主要差别在于定义层面：前面我们几乎都在单元层面工作，这次将稍微上移，将TDD应用于功能测试和集成测试。

我们选择的框架为JBehave和Selenide。

本章涵盖如下主题：

- 不同文档；
- 行为驱动开发；
- 书店应用程序的BDD故事；
- JBehave。

7.1 不同规范

前面说过，TDD的好处之一是，可执行文档始终是最新的。然而，仅有通过单元测试获得的文档还不够；在单元这个很低的层面工作时，我们对细节洞若观火，但很容易“只见树木，不见

森林”。例如，如果我们审视为“井字游戏”创建的规范，很可能领会不到这个应用程序的要点：你明白每个单元的功能，也知道它是如何与其他单元互操作的，却难以掌握其背后的理念。更准确地说，你知道单元X做的事情是Y，并与单元Z通信，却难以找到功能文档及其背后的理念。

对开发来说亦是如此。要着手处理表现为单元测试的规范，我们需要对系统有更全面的了解。本书前面都先列出需求，再根据需求编写规范，然后根据规范编写实现。随后，这些需求都被丢弃，杳无音信。我们没有将它们放入仓库，也没有根据它们验证工作成果。

7.1.1 文档

我们合作过的很多组织中，创建文档的出发点就是错误的。管理层通常认为文档与项目成败有一定关系，如果没有大量（通常是短命的）文档，项目就会失败。因此，大家被要求花大量时间进行规划、回答问题以及填写问卷，而这些问卷的设计常常对项目毫无帮助，却旨在营造“一切尽在掌握”的假象。有些职位就是为文档而存在的——这个文档就是我的工作成果。文档还被作为安慰剂，让人觉得一切都在按计划进行——有Excel表指出了这一点。然而，创建文档最常见的原因是，有规章制度要求必须创建某些文档；即便你怀疑这些文档的价值，但规章制度神圣不可侵犯，你必须创建。

文档不仅可能以错误的目的创建，从而不能提供足够的价值，而且常常可能有巨大的破坏力。既然创建了文档，自然要信任它，可如果文档不是最新的，结果将如何呢？需求在变化，有bug得到修复，有新功能被开发，还有一些既有功能被删除。

只要时间足够长，所有传统文档都将过时。每次修改代码都更新文档是项庞大而复杂的任务，因此迟早会出现静态文档没有反映实际情况的问题。如果我们完全信任不准确的东西，开发工作基于的假设就将是错误的。

唯一准确的文档是我们编写的代码。代码是我们开发的，也是我们部署的，只有它们才能完全准确反映应用程序。然而，并非参与项目的每个人都能读懂代码，除程序员外，我们还可能需要与管理人员、测试人员、业务人员、最终用户等合作。

为更准确地确定什么样的文档更好，下面深入研究潜在的文档使用者。为简单起见，我们将这些文档使用者分为两类：程序员（能够阅读并理解代码的使用者）和非程序员（其他使用者）。

7.1.2 供程序员使用的文档

开发人员与代码打交道，鉴于我们已确定代码是最准确的文档，因此没有理由不使用它们。如果你要搞明白某个方法是做什么的，请看这个方法的代码。对某个类的作用心存疑惑？请看这个类的代码。有一段代码看不明白？这就麻烦了！然而，导致这种麻烦的不是文档缺失，而是代

码本身写得不好。

仅通过研究以理解代码通常还不够。即便你明白了代码是做什么的，其目的也可能不明显：最初为何要编写这些代码呢？

此时，规范便可派上用场。我们不仅不断使用规范以验证代码，它们还是可执行的文档。规范总是最新的，否则它们执行时就会失败。另外，虽然代码应该编写得易于阅读和理解，但帮助理解某段实现代码的编写原因、逻辑和动机方面，规范提供了更容易的捷径。

将代码作为文档并不排斥其他类型的文档。恰恰相反，关键不是避免使用静态文档，而是避免重复。代码提供了必要的细节时，优先查看代码而不是其他文档；大多数情况下，其他文档指的是较简略的文档，如概述、系统的总体目标、使用的技术、环境搭建、安装、构建、打包以及其他类型的数据。它们不提供详细信息，更像是指南和快速入门。对于这些文档，markdown格式（http://whatismarkdown.com/）的简单README文件通常是最佳的。

对于基于代码的文档，TDD是最佳的创建方式。到目前位置，我们都是在单元（方法）级工作，还没有在更高层面（如功能规范）应用TDD，但这样做之前，先看看团队中的其他角色。

7.1.3　供非程序员使用的文档

传统测试人员通常与开发人员归属于完全不同的小组，这种划分导致更多测试人员不熟悉代码，并认为自己做的是质量检查工作。他们是流程末端的检验员，像边防警察一样决定哪些代码可以部署、哪些需要返工。另一方面，越来越多的组织将测试人员视为开发团队的一员，其职责是确保软件的内生质量。这类测试人员必须熟悉代码，对他们来说，将代码作为文档再自然不过。然而，对于第一类测试人员（即不懂代码的测试人员），该如何办呢？另外，并非只有一些测试人员不懂代码，管理人员、最终用户、业务代表等也不懂代码。无法阅读并理解代码的人到处都是。

我们需要寻找一种方式保留可执行的文档提供的优点，同时以人人都能理解的方式编写。另外，在TDD中，应该从一开始创建可执行文档时，就让所有人都参与：让他们定义用于开发应用程序和验证开发成果的需求。我们需要指定将做什么的简略规范，因为详尽规范由单元测试提供。总之，我们需要这样的文档：可作为需求，可执行，可对工作进行验证，人人都能编写和理解。

下面介绍行为驱动开发。

7.2　行为驱动开发

BDD是一种敏捷过程，旨在项目开发过程中始终专注于相关方的利益。这是一个TDD变种，它也预先定义规范，根据规范完成实现并定期运行规范以验证结果。此外，还有一些不同之处。

不像TDD那样基于单元测试，BDD倡导编写多个规范（称为场景）再开始实现（编码）。虽然没有具体的规则，但BDD通常偏重于简约的功能需求。BDD虽然可用于单元层面，但仅当采用人人都能编写和理解的简略方法时，才能得到实际好处。另一个不同之处是受众：BDD适用于所有人——程序员、测试人员、管理人员、最终用户、业务代表等。TDD基于单元层级，可以说是从内到外的（从单元开始，逐步延伸到功能），而BDD通常被认为是从外到内的（从功能着手，往内逐步延伸到单元）。行为驱动开发扮演的是验收标准和就绪程度指示器的角色，指出产品何时完工并可部署到生产环境。

我们首先定义功能（或行为），再使用TDD和单元测试实现；实现完整的行为后，再使用BDD验证。一个BDD场景可能需要几小时乃至几天才能完成，在此期间，我们可使用TDD和单元测试。完成后，运行BDD场景做最后的验证。TDD是针对程序员的，周期非常短；而BDD针对所有人，周期要长得多。对于每个BDD场景，都有大量TDD单元测试。

此时你可能一头雾水，不知道BDD到底为何物。有鉴于此，下面对其进行详细介绍，首先解释其样式。

7.2.1 叙述

BDD故事由叙述（narrative）和至少一个紧跟其后的场景组成。叙述只起说明作用，其主要用途是提供足够的信息，为参与各方（测试人员、业务代表、开发人员、分析人员等）深入交流打基础。叙述简短地描述一个功能，这种描述是从要求提供该功能的人的角度进行的。

叙述的目标是回答如下三个基本问题：

(1) 要开发的功能的好处或价值（In order to）？
(2) 谁需要这项功能（As a）？
(3) 要开发什么样的功能（I want to）？

回答这些问题后，开始着手定义我们认为的最佳解决方案。这种思维过程的结果是提供详细信息的场景。

目前为止，我们都在非常低的层级工作，并将单元测试作为驱动力。我们从程序员的角度指定要创建什么；假定高级需求已定义好，而我们的职责是根据需求编写代码。现在回到开头，变身为客户或业务代表：有人出了一个绝妙的点子，而我们正与团队的其他成员讨论这个点子。简而言之，我们想打造一个网上书店；当前还处于设想阶段，我们甚至都不确定将如何开发，因此想开发一个**最小可行产品**（**MVP**）。我们要研究的一个角色是书店管理员，他应能够添加新书以及更新或删除既有图书。所有这些操作都应该是可执行的，因为我们希望管理员能够高效管理书店的图书。对于这个角色，我们提出的叙述如下：

```
为了高效管理书店
作为书店管理员
希望能够添加、更新、删除图书
```

至此，我们知道了好处是什么（高效管理图书）、谁需要（管理员）以及需要开发什么样的功能（插入、更新和删除操作）。别忘了，这并非有关我们该做什么的详尽描述。叙述旨在发起讨论，这种讨论的结果是一个或更多场景。

不同于TDD单元测试，叙述以及BDD故事的其他部分是任何人都能够编写的。它们不要求编写者有编程技能，也不需要太详细。根据所在的组织，可能由同一个人（业务代表、产品所有者、客户等）编写所有叙述，也可能由整个团队协作编写。

对叙述有更清晰的认识后，下面看看场景。

7.2.2　场景

叙述旨在促进交流，而场景是交流的结果。场景应描述（As a部分指定的）角色与系统的交互。单元测试是开发人员编写的代码，供开发人员使用；BDD场景与此不同，它们应使用平实的语言定义，并包含尽可能少的技术细节，让项目的所有参与者（开发人员、测试人员、设计人员、管理人员、客户等）能针对要在系统中添加哪些行为（或功能）达成一致。

场景是叙述的验收标准。只要与叙述相关的场景都能成功运行，就可认为工作完成。场景与单元测试很像，主要差别在于涵盖的范围（一个方法还是整个功能）以及实现所需的时间（几秒钟、几分钟还是几小时乃至几天）。与单元测试类似，场景也用于驱动开发，是在开发之前定义的。

每个场景都包含一个描述，还有一个或多个以Given、When或Then开始的步骤。描述很短，只是为了提供一些信息，让我们能够迅速明白场景的目的。另一方面，步骤是场景的一系列前置条件、事件和期望结果；它们帮助我们明确定义行为，可轻松转换为自动化测试。

本章将更多专注于BDD的技术方面及其对开发人员心态的影响。有关BDD更广泛的用途和更深入的讨论，参阅Gojko Adzic的著作《实例化需求：团队如何交付正确的软件》。

Given步骤定义了上下文，即为让场景的其他部分获得成功而需要满足的前置条件。根据前面有关图书管理的叙述，一个这样的前置条件可能如下所示：

```
Given 用户当前在图书页面
```

这个前置条件很简单，但必不可少。我们的网站可能有很多页面，因此执行任何操作前，都需确保用户位于正确的页面。

When步骤定义了操作或某种事件。前面的叙述中，我们指定了管理员必须能够添加、更新

和删除图书。下面看看与删除相关的操作：

```
When 用户选择一本书
When 用户点击删除按钮
```

这个示例中，我们使用When步骤定义了多个操作：先选择一本书，再单击按钮Delete the book。在这里，我们指定要单击的按钮时，使用的是ID（deleteBook）而不是文本（Delete the book）。大多数情况下，使用ID更好，因为ID有多个优点：它们是独一无二的（在给定页面中，每个ID只能出现一次）；它们向开发人员发出了更明确的指令（创建一个ID为deleteBook的元素）；它们不受当前页面上其他变化的影响。元素的文本很容易发生变化；而元素文本发生变化后，所有使用它的场景也将失败。就网站而言，另一种指定元素的方式是使用XPath，但应尽可能避免这样做。因为只要HTML结构发生细微的变化，指定的XPath就不再正确。

与单元测试类似，场景也应是可靠的，并在功能没有实现或发生问题时失败。否则，当它们出现漏报问题后，大家自然就会对规范置若罔闻。

最后，我们应该总是在场景末尾包含验证，应该指定操作的期望结果。在前面的场景中，Then步骤可能如下所示：

```
Then 图书已被删除
```

这里定义的结果做了很好的折衷：既提供了足够的信息，又未涉及设计细节。我们原本可以提及数据库，甚至具体到MongoDB数据库。然而，很多情况下，这种信息从行为角度看都不重要。我们应只确认图书从目录中删除了，而不管它存储在什么地方。

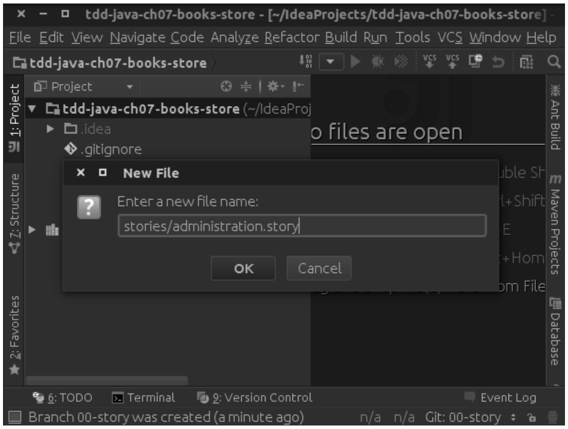
熟悉BDD故事的格式后，下面编写书店应用程序的BDD故事。

7.3　书店应用程序的 BDD 故事

首先，复制https://bitbucket.org/vfarcic/tdd-java-ch07-books-store处的代码。这是一个空项目，本章将始终使用它。与前几章一样，每节都在这个仓库中有对应的分支，以免遗漏。

我们将编写一个BDD故事，它是纯文本的，使用的是普通英语，且不包含任何代码。这样，所有利益相关方都能参与，而不管其编码水平如何。本章后面将演示如何自动化这个故事。

先在目录stories中新建一个`administration.story`文件。



前面已经编写了叙述，因此我们将接着这个叙述往下编写。

```
叙述：
为了高效管理书店
作为书店管理员
希望能够添加、更新、删除图书
```

我们将使用JBehave格式编写故事。有关JBehave的更多细节将稍后介绍，届时请访问http://jbehave.org/获取更多信息。

叙述都以`Narrative`行打头，后面跟着`In order to`、`As a`和`I want to`行，这几行的含义前面已经讨论过。

知道为何、是谁和什么后，该与团队的其他人员一起坐下来讨论可能的场景了。此时我们不讨论步骤（`Given`、`When`和`Then`），而只对潜在场景做简要描述。这些场景可能是如下所示：

```
Scenario：图书细节表单应当拥有全部字段
Scenario：用户应当可以创建一本新书
Scenario：用户应当可以展示图书细节
Scenario：用户应当可以更新图书细节
Scenario：用户应当可以删除图书
```

这里使用的是JBehave语法：先写出单词Scenario，再做简短的描述。这个阶段没有理由涉及细节，旨在进行快速头脑风暴。在这里，我们想出了5个场景，其中第一个定义了将用于管理图书的表单字段，而其他场景试图定义各种管理任务。这些场景真的没什么创意可言。我们要做的是开发一个非常简单的应用程序的MVP，如果结果表明这个产品很成功，我们就可发挥创意对其进行扩展。就当前目标而言，这个应用程序将十分简单直白。

知道场景都是什么后，该对它们做正确而简要的定义了。从第一个场景着手：

```
Scenario：图书细节表单应当拥有全部字段
```

```
Given 用户当前在图书页面
Then 字段bookId存在
Then 字段bookTitle存在
Then 字段bookAuthor存在
Then 字段bookDescription存在
```

这个场景没有包含任何操作——没有When步骤，可将其视为完整性检查。它告诉开发人员，图书表单应包含哪些字段。根据这些字段，可确定要使用的数据模式（schema）。这些ID的描述性足够强，让我们知道每个字段都是做什么的（一个图书ID和三个文本字段）。别忘了，这个场景（以及后续所有场景）都是纯文本的，不包含任何代码。这种场景的主要优点是谁都能够编写，我们将尽力坚持这种做法。

下面看看第二个场景：

```
Scenario：用户应当可以创建一本新书

Given 用户当前在图书页面
When 用户点击newBook按钮
When 用户为图书表单设置值
When 用户点击saveBook按钮
Then 保存图书
```

这个场景比前一个更正规，其中有一个明确的前置条件（用户访问的是特定页面），有多个操作（单击按钮**newBook**、填写表单并单击按钮**saveBook**），还有结果验证（保存图书）。

其他场景如下（这些场景都与前面的场景类似，因此没有必要分别解释）：

```
Scenario：用户应当可以展示图书细节

Given 用户当前在图书页面
When 用户选择一本书
Then 图书表单包含所有数据

Scenario：用户应当可以更新图书细节

Given 用户当前在图书页面
When 用户选择一本书
When 用户为图书表单设置值
Then 保存图书

Scenario：用户应当可以删除图书

Given 用户当前在图书页面
When 用户选择一本图书
When 用户点击deleteBook按钮
Then 删除图书
```

唯一需要指出的一点是，我们在合适的情况下使用了相同步骤（如When用户选择一本图书）。稍后将尝试自动化所有场景，通过在相同步骤中使用相同文本，可避免复制代码，从而节省时间。我们必须在以最佳方式自由表达场景和简化自动化之间取得平衡，这很重要。这些场景还有一些

可修改的地方，但对它们进行重构前，先简要介绍JBehave。

可在Git仓库tdd-java-ch07-books-store的分支00-story（https://bitbucket.org/vfarcic/tdd-java-ch07-books-store/branch/00-story）中找到源代码。

7.4　JBehave

要让JBehave能够运行BDD故事，必须有两个主要的组件：运行器和步骤。运行器是一个类，它对故事进行分析，运行所有场景并生成报告；步骤则是与场景中步骤匹配的代码方法。本章的项目已经包含所有Gradle依赖，因此可以直接创建JBehave运行器。

7.4.1　JBehave 运行器

每种类型的测试都需要一个运行器，JBehave中也不例外。前几章中，我们使用了JUnit和TestNG运行器。这些框架中都不需要做特殊配置，但JBehave的更苛刻，要求我们必须创建一个类，用于存储运行故事所需的所有配置。

下面是贯穿本章都将使用的Runner类的代码：

```
public class Runner extends JUnitStories {

    @Override
    public Configuration configuration() {
        return new MostUsefulConfiguration()
                .useStoryReporterBuilder(getReporter())
                .useStoryLoader(new LoadFromURL());
    }

    @Override
    protected List<String> storyPaths() {
        String path = "stories/**/*.story";
        return new StoryFinder().findPaths(
            CodeLocations
                    .codeLocationFromPath("")
                    .getFile(),
            Collections
                    .singletonList(path),
            new ArrayList<String>(),
            "file:"
        );
    }

    @Override
    public InjectableStepsFactory stepsFactory() {
        return new InstanceStepsFactory(
                configuration(),
                new Steps()
```

```
        );
    }

    private StoryReporterBuilder getReporter() {
        return new StoryReporterBuilder()
            .withPathResolver(
                new FilePrintStreamFactory
                        .ResolveToSimpleName()
            )
            .withDefaultFormats()
            .withFormats(Format.CONSOLE, Format.HTML);
    }

}
```

这些代码很平常，因此只解释其中的几个重要部分。重写的方法storyPaths将故事文件的位置设置为stories/**/*.story，这是一种标准的Apache Ant（http://ant.apache.org/）语法。用通俗的语言说，这意味着包含目录stories及其所有子目录（**）中所有扩展名为.story的文件。另一个重要的重写方法是stepsFactory，它用于设置包含步骤定义的类（我们马上就会编写它）。在这里，我们将其设置为**Steps**类的实例（仓库包含一个我们将在后面使用的空类）。

可在Git仓库tdd-java-ch07-books-store的分支01-runner（https://bitbucket.org/vfarcic/tdd-java-ch07-books-store/branch/01-runner）中找到源代码。

编写运行器代码后，启动并查看结果。

7.4.2 待定步骤

可使用下面的Gradle命令运行场景：

```
$ gradle clean test
```

Gradle只运行前一次执行后修改过的任务。由于源代码变化不频繁（我们通常只修改文本格式的故事），因此运行test任务前，必须运行clean任务将缓存删除。

JBehave创建了一个不错的报告，并将其放在目录target/jbehave/view中。请使用你喜欢的浏览器打开这个目录中的文件reports.html。

报告的第一页列出了我们定义的故事（这里只有故事Administration），还有两个预定义的故事——**BeforeStories**和**AfterStories**。这两个故事的用途与JUnit中用@BeforeClass和@AfterClass注解的方法类似：在故事之前和之后运行，可用于设置和拆除数据、服务器等。

报告的第一页表明，我们有5个场景，它们都处于待定（Pending）状态。JBehave通过这种方式告诉我们，场景既未成功也未失败，而是使用过的步骤缺失代码。

Story Reports

Stories		Scenarios				
Name	Excluded	Total	Successful	Pending	Failed	Excluded
Administration	0	5	5	5	0	0
AfterStories	0	0	0	0	0	0
BeforeStories	0	0	0	0	0	0
3	**0**	**5**	**5**	**5**	**0**	**0**

每行的最后一列都包含一个链接，让我们能够查看相应故事的详情。

Narrative:

In order to manage the book store collection
As a store administrator
I want to be able to perform insert, update and delete operations

Scenario: Book details form should have all fields

Given user is on the books screen　(PENDING)
Then field bookId exists　(PENDING)
Then field bookTitle exists　(PENDING)
Then field bookAuthor exists　(PENDING)
Then field bookDescription exists　(PENDING)

```
@Given("user is on the books screen")
@Pending
public void givenUserIsOnTheBooksScreen() {
  // PENDING
}
```

在这里，所有步骤都被标记为“待定”，JBehave甚至提出了建议，指出需要为每个待定步骤创建一个方法。

目前为止，我们编写一个包含5个场景的故事。每个场景都相当于一个规范，用于指定我们应开发什么以及验证是否正确完成了开发。这些场景都包含多个定义前置条件（`Given`）、操作（`When`）和期望结果（`Then`）的步骤。

现在编写步骤背后的代码，但开始前，先介绍Selenium和Selenide。

7.4.3　Selenium 和 Selenide

Selenium是一组用于自动化浏览器的驱动程序，我们可使用它们操作浏览器和页面元素，如单击按钮或链接、填写表单字段、打开特定的URL等。几乎有用于任何浏览器的驱动程序：

Android、Chrome、FireFox、Internet Explorer、Safari等等。我们喜欢的是PhantomJS，这是一款无界面浏览器（headless browser），不使用任何UI就能工作。使用它运行故事比使用传统浏览器要快，我们常用其快速获取有关Web应用程序就绪程度的反馈。如果Web应用程序像预期那样工作，就可接着尝试在要支持的所有浏览器和版本中运行。

有关Selenium的更详细信息，请参阅http://www.seleniumhq.org/；有关它支持的驱动程序清单，请参阅http://www.seleniumhq.org/projects/webdriver/。

虽然Selenium非常适合用于自动化浏览器，但它也有缺点，其中之一就是在很低的层面操作。例如，单击按钮很容易，只需使用一行代码就能实现：

```
selenium.click("myLink")
```

如果ID为`myLink`的元素不存在，Selenium将引发异常，测试将失败。虽然我们希望测试在"期望的元素不存在"时失败，但很多情况下，情况并非这样简单。例如，页面可能是动态加载的，期望的元素仅在向服务器发出的异步请求得到响应后才出现。有鉴于此，我们可能希望等到这个元素出现后再单击——仅当等待超时时，测试才失败。虽然使用Selenium能够实现这样的目标，但既繁琐又容易出错。另外，对于Selenide已经做了的工作，我们为何还要去做呢？下面介绍Selenide。

Selenide（http://selenide.org/）是一个Selenium WebDriver包装器，其API更简洁，还支持Ajax、jQuery式选择器等。对于所有Web步骤，我们都将使用Selenide，稍后你将更深入地了解它。

下面编写一些代码。

7.4.4 JBehave 步骤

着手编写步骤前，先安装PhantomJS浏览器。有关如何在你使用的操作系统中安装这个浏览器的说明，请参阅http://phantomjs.org/download.html。

安装`PhantomJS`后，指定几个Gradle依赖：

```
dependencies {
    testCompile 'junit:junit:4.+'
    testCompile 'org.jbehave:jbehave-core:3.+'
    testCompile 'com.codeborne:selenide:2.+'
    testCompile 'com.codeborne:phantomjsdriver:1.+'
}
```

你对JUnit和jbehave-core很熟悉，这两个依赖都在前面指定过；这里新增的两个依赖是Selenide和PhantomJS。请刷新Gradle依赖，将它们包含到你的IDEA项目。

将PhantomJS WebDriver添加到`Steps`类：

```
public class Steps {

    private WebDriver webDriver;

    @BeforeStory
    public void beforeStory() {
        if (webDriver == null) {
            webDriver = new PhantomJSDriver();
            WebDriverRunner.setWebDriver(webDriver);
            webDriver.manage().window().setSize(
                    new Dimension(1024, 768)
            );
        }
    }

}
```

我们使用注解`@BeforeStory`指定了进行基本设置的方法。如果没有指定驱动程序，我们就将其设置为`PhantomJSDriver`。由于这个应用程序在小型设备（手机、平板电脑等）上的外观不同，因此必须明确指定屏幕大小，这很重要。这里，我们将其设置为台式机/笔记本电脑显示器的适中屏幕分辨率——1024 × 768。

完成设置后，编写第一个待定步骤的代码。为此，可以直接复制`JBehave`在报告中推荐的第一个方法：

```
@Given("user is on the books screen")
public void givenUserIsOnTheBooksScreen() {
    // PENDING
}
```

想象我们的应用程序包含一个打开图书页面的链接。为打开这个页面，需要执行两个步骤：

(1) 打开网站主页；
(2) 单击菜单中的books链接。

我们将这个链接的ID指定为`books`。ID非常重要，使我们能够轻松找到页面中的元素。

可将前述步骤转换为如下代码：

```
private String url = "http://localhost:9001";

@Given("user is on the books screen")
public void givenUserIsOnTheBooksScreen() {
    open(url);
    $("#books").click();
}
```

这里假设我们的程序将运行于本地主机（localhost）的9001端口，因此首先打开主页URL，再单击ID为books的元素（指定ID的`Selenide/jQuery`语法为`#`）。

如果再次运行前面定义的运行器，将看到第一个步骤失败，而其他步骤依然处于待定状态。此时，我们处于“红灯–绿灯–重构”周期的红灯阶段。

接着处理第一个场景使用的其他步骤。第二个步骤的代码可能如下所示：

```
@Then("field bookId exists")
public void thenFieldBookIdExists() {
    $("#books").shouldBe(visible);
}
```

第三个步骤几乎与此相同，因此可以重构前一个方法，将元素ID转换为变量：

```
@Then("field $elementId exists")
public void thenFieldExists(String elementId) {
    $("#" + elementId).shouldBe(visible);
}
```

经过这样的修改后，第一个场景中的步骤就都完成了。如果再次运行测试，结果将如下所示。

Scenario: Book details form should have all fields

Given user is on the books screen (FAILED)
Element not found {#books} Expected: visible Screenshot: file:/home/vfarcic/IdeaProjects/tdd-java-ch07-books-store/build/reports/tests/1430688921325.15.png Timeout: 4 s. Caused by: NoSuchElementException: Error Message => 'Unable to find element with css selector '#books''

Then field bookId exists (NOT PERFORMED)

Then field bookTitle exists (NOT PERFORMED)

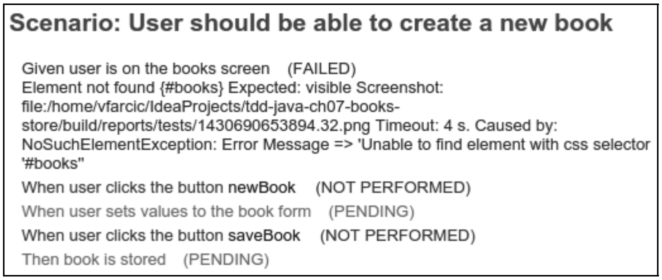
Then field bookAuthor exists (NOT PERFORMED)

Then field bookDescription exists (NOT PERFORMED)

第一个步骤失败了，因为我们还未开始编写书店应用程序的实现。Selenide有一项很不错的功能：每当失败时都创建浏览器的屏幕截图，并在报告中指出其存储路径。其他步骤都处于未执行（not performed）状态，因为失败后场景便会停止执行。

接下来该做什么呢？这取决于团队的结构。如果功能测试和实现由同一人负责，这个人就可开始处理实现——编写刚好让这个场景能够通过的代码。功能测试和实现代码通常由不同的人负责，这种情况下，负责功能测试的人可接着编写其他场景中缺失的步骤，而负责实现代码的人可着手处理实现。由于所有场景都以文本方式编写，程序员知道该做什么，因此他和负责功能测试的人可并行工作。这里假设属于后一种情况，因此接着为其他待定步骤编写代码。

下面运行第二个场景。



Scenario: User should be able to create a new book

Given user is on the books screen (FAILED)
Element not found {#books} Expected: visible Screenshot:
file:/home/vfarcic/IdeaProjects/tdd-java-ch07-books-store/build/reports/tests/1430690653894.32.png Timeout: 4 s. Caused by:
NoSuchElementException: Error Message => 'Unable to find element with css selector '#books''

When user clicks the button newBook (NOT PERFORMED)

When user sets values to the book form (PENDING)

When user clicks the button saveBook (NOT PERFORMED)

Then book is stored (PENDING)

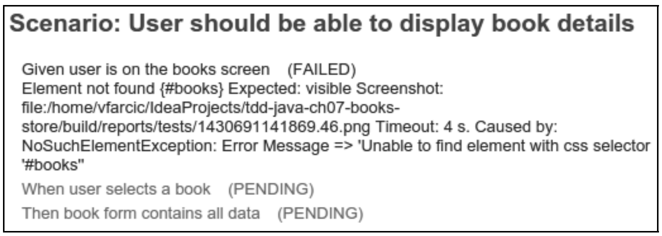
这个场景中，一半的步骤已在前一个场景完成，只有两个步骤待定。单击按钮**newBook**后，我们应该在表单中填写一些值，再单击按钮**saveBook**，并验证图书被正确保存。对于最后一部分，可通过检查图书是否出现在可售图书列表完成。

缺失的步骤可能如下所示：

```
@When("user sets values to the book form")
public void whenUserSetsValuesToTheBookForm() {
    $("#bookId").setValue("123");
    $("#bookTitle").setValue("BDD Assistant");
    $("#bookAuthor").setValue("Viktor Farcic");
    $("#bookDescription").setValue(
        "Open source BDD stories editor and runner"
    );
}
@Then("book is stored")
public void thenBookIsStored() {
    $("#book123").shouldBe(present);
}
```

第二个步骤假定每本可售图书都有一个格式为`book[ID]`的ID。

来看下一个场景。



Scenario: User should be able to display book details

Given user is on the books screen (FAILED)
Element not found {#books} Expected: visible Screenshot:
file:/home/vfarcic/IdeaProjects/tdd-java-ch07-books-store/build/reports/tests/1430691141869.46.png Timeout: 4 s. Caused by:
NoSuchElementException: Error Message => 'Unable to find element with css selector '#books''

When user selects a book (PENDING)

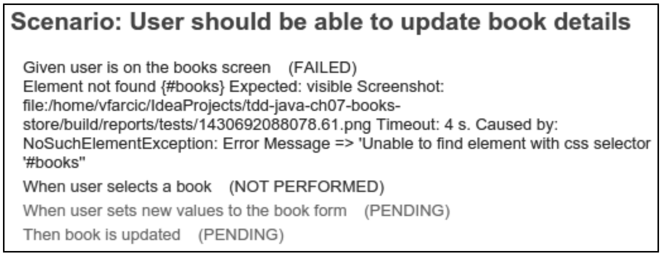
Then book form contains all data (PENDING)

与前一个场景一样，有两个待定步骤需要开发。我们需要一种选择图书的方式，还需验证表单中的数据是否正确：

```
@When("user selects a book")
public void whenUserSelectsABook() {
    $("#book1").click();
}
```

```
@Then("book form contains all data")
public void thenBookFormContainsAllData() {
    $("#bookId").shouldHave(value("1"));
    $("#bookTitle").shouldHave(
        value("TDD for Java Developers")
    );
    $("#bookAuthor").shouldHave(value("Viktor Farcic"));
    $("#bookDescription").shouldHave(value("Cool book!"));
}
```

这两个方法很有趣，因为它们不仅指定了期望的行为（特定图书链接被单击时，将显示一个包含其数据的表单），还期望有一些可用于测试的数据。这个场景运行时，应存在这样的图书：ID为1、书名为TDD for Java Developers、作者为Viktor Farcic、描述为Cool book!。我们可将这些数据添加到数据库，也可使用模拟服务器提供这些预定义的值。无论我们如何设置这些测试数据，都可完成这个场景并进入下一个。



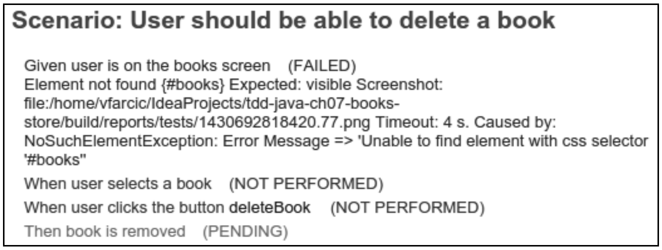
这些待定步骤的实现可能如下所示：

```
@When("user sets new values to the book form")
public void whenUserSetsNewValuesToTheBookForm() {
    $("#bookTitle").setValue(
        "TDD for Java Developers revised"
    );
    $("#bookAuthor").setValue(
        "Viktor Farcic and Alex Garcia"
    );
    $("#bookDescription").setValue("Even better book!");
    $("#saveBook").click();
}

@Then("book is updated")
public void thenBookIsUpdated() {
    $("#book1").shouldHave(
        text("TDD for Java Developers revised")
    );
    $("#book1").click();
    $("#bookTitle").shouldHave(
        value("TDD for Java Developers revised")
    );
    $("#bookAuthor").shouldHave(
```

```
            value("Viktor Farcic and Alex Garcia")
        );
        $("#bookDescription").shouldHave(
            value("Even better book!")
        );
    }
```

至此，只余下一个场景。


Scenario: User should be able to delete a book

Given user is on the books screen (FAILED)
Element not found {#books} Expected: visible Screenshot: file:/home/vfarcic/IdeaProjects/tdd-java-ch07-books-store/build/reports/tests/1430692818420.77.png Timeout: 4 s. Caused by: NoSuchElementException: Error Message => 'Unable to find element with css selector '#books''
When user selects a book (NOT PERFORMED)
When user clicks the button deleteBook (NOT PERFORMED)
Then book is removed (PENDING)

可通过确认图书不包含于可售图书列表验证它已被删除：

```
@Then("book is removed")
public void thenBookIsRemoved() {
    $("#book1").shouldNotBe(visible);
}
```

至此，步骤的代码全部编写完成。现在，开发应用程序的人不仅有需求，还有验证每个行为（场景）的途径，可使用“红灯–绿灯–重构”周期每次实现一个场景。

可在Git仓库`tdd-java-ch07-books-store`的分支`02-steps`（https://bitbucket.org/vfarcic/tdd-java-ch07-books-store/branch/02-steps）中找到源代码。

7.4.5 最后的验证

假设有另一人负责编写代码以实现场景指定的需求，它每次挑选一个场景，开发代码并运行场景以确认其实现是正确的。所有场景都实现后，运行整个故事进行最后的验证。

我们将这个应用程序打包为一个`Docker`文件，并创建运行该应用程序的Vagrant虚拟机。

请签出分支https://bitbucket.org/vfarcic/tdd-java-ch07-books-store/branch/03-validation并运行`Vagrant`：

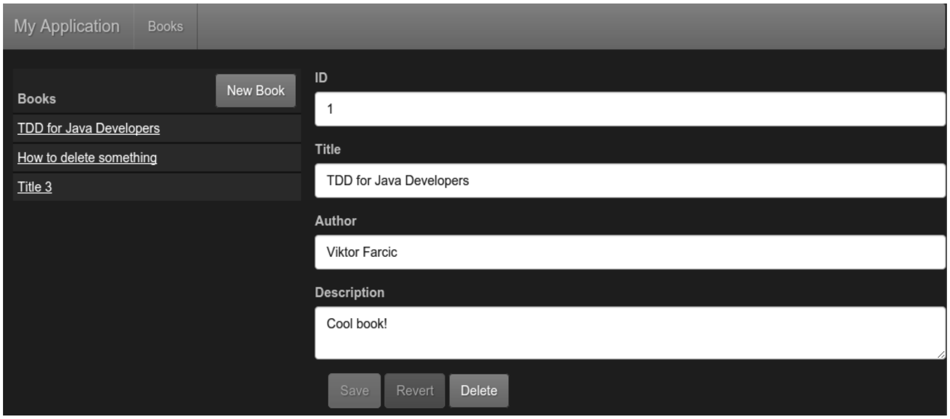
```
$ vagrant up
```

输出应如下所示：

```
==> default: Importing base box 'ubuntu/trusty64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/trusty64' is up to date...
```

```
…
==> default: Running provisioner: docker...
    default: Installing Docker (latest) onto machine...
    default: Configuring Docker to autostart containers...
==> default: Starting Docker containers...
==> default: -- Container: books-fe
```

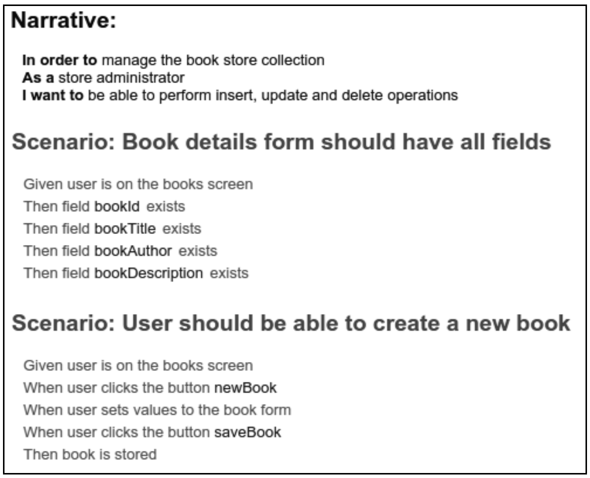
Vagrant虚拟机安装完毕后，可在浏览器中输入http://localhost:9001运行这个应用程序：



现在再次运行我们的场景：

```
$ gradle clean test
```

这次没有失败，所有场景都成功运行。



所有场景都通过后，满足验收标准，可将应用程序部署到生产环境。

7.5　小结

从本质上说，BDD是一个TDD变种，遵循同样的基本原则：先编写测试（场景），再编写实现代码。它驱动开发，并帮助我们更好地理解该做什么。

主要差别之一是周期的持续时间。基于单元测试的TDD中，我们快速从红灯切换到绿灯（在几分钟乃至几秒钟内）；而BDD通常采用更高级的方法，从红灯切换到绿灯可能需要几小时乃至几天。另一个重要差别是受众。基于单元测试的TDD是开发人员为开发人员做的；而BDD使用的是普通语言，通常涉及所有团队成员。

有关这个主题可以写部专著，而我们的目的是向你提供足够信息，让你能够更深入地研究BDD。

现在看看遗留代码，以及如何对遗留代码进行修改，使其对测试驱动开发更友好。

第 8 章

重构遗留代码——使其重焕青春

8

“恐惧是通向黑暗之路。恐惧导致愤怒；愤怒引发仇恨；仇恨造成痛苦。”

——尤达

TDD不能直接应用于遗留代码，可能需要稍微调整步骤才管用。用于处理遗留代码时，TDD可能需要调整，即你执行的不再是你习惯的TDD。本章带你进入遗留代码领域，并尽可能吸收TDD的养分。

我们将改弦易辙，处理一个正用于生产的遗留应用程序。我们将小步修改这个应用程序，以免导致缺陷或退化，这样就能早早吃上午餐！

本章涵盖如下主题：

- 遗留代码；
- 处理遗留代码；
- REST通信；
- 依赖注入；
- 不同层级的测试：端到端测试、集成测试和单元测试。

8.1 遗留代码

先定义“遗留代码”。很多作者都给“遗留代码”下了不同定义，如受信任的应用程序或测试、不再支持的代码等，但我们最喜欢Michael Feathers的定义：

“遗留代码就是不带测试的代码。

为什么这样定义呢？因为它是客观的：要么带测试，要么不带测试。”

——Michael Feathers

如何识别遗留代码呢？虽然遗留代码通常都充斥着糟糕的代码，但Michael Feathers在其著作《修改代码的艺术》中指出了遗留代码的一些坏味。

> **代码坏味**
>
> 坏味是代码中违背了基本设计元素并给设计质量带来负面影响的结构。代码坏味通常不同于bug，它们从技术上说是正确的，不会导致程序无法正常运行。而是昭示着设计存在缺陷，这些缺陷可能影响开发速度或增加未来出现bug或故障的风险。
>
> ——摘自http://en.wikipedia.org/wiki/Code_smell

遗留代码存在的一种常见坏味是无法测试。它们访问外部资源、带来其他副作用、使用new运算符等。一般而言，良好的设计都易于测试。下面看一些遗留代码。

遗留代码示例

要对软件概念进行诠释，通常最容易的方式是通过代码，这个概念也不例外。第3章介绍并编写了“井字游戏”，其中的如下代码检查落子位置是否有效：

```
public class TicTacToe {

    public void validatePosition(int x, int y) {
        if (x < 1 || x > 3) {
            throw new RuntimeException("X is outside "
              + "board");
        }
        if (y < 1 || y > 3) {
            throw new RuntimeException("Y is outside "
              + "board");
        }
    }
}
```

与这些代码对应的规范如下所示：

```
public class TicTacToeSpec {

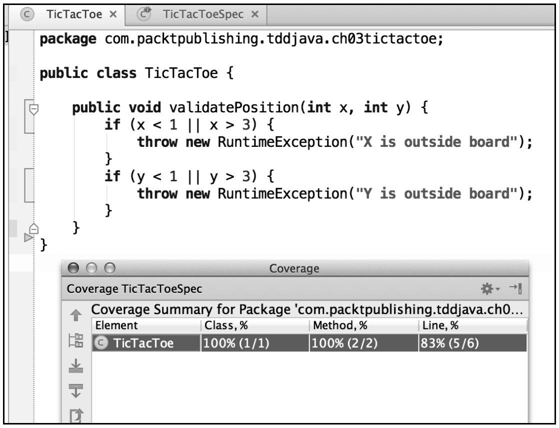
    @Rule
    public ExpectedException exception =
      ExpectedException.none();
    private TicTacToe ticTacToe;

    @Before
    public final void before() {
        ticTacToe = new TicTacToe();
    }
```

```
    @Test
    public void whenXOutsideBoardThenRuntimeException()
    {
        exception.expect(RuntimeException.class);
        ticTacToe.validatePosition(5, 2);
    }

    @Test
    public void whenYOutsideBoardThenRuntimeException()
    {
        exception.expect(RuntimeException.class);
        ticTacToe.validatePosition(2, 5);
    }
}
```

JaCoCo报告指出，测试覆盖了除最后一行（结束方法的大括号）外的所有代码。



确信测试很好地覆盖了代码后，就可凭感觉进行安全的重构了（片段）：

```
public class TicTacToe {

    public void validatePosition(int x, int y) {
        if (isOutsideTheBoard(x)) {
            throw new RuntimeException("X is outside "
              + "board");
        }
        if (isOutsideTheBoard(y)) {
            throw new RuntimeException("Y is outside "
              + "board");
        }
    }

    private boolean isOutsideTheBoard
      (final int position) {
        return position < 1 || position > 3;
    }
}
```

8

这些代码应该没问题，因为测试成功，且测试的代码覆盖率很高。

你可能也这样认为，但有一点需要注意：对于`RuntimeException`块中的消息，我们并未检查其正确性，虽然代码覆盖率报告指出“覆盖了所有分支”。

> **代码覆盖率到底是什么？**
>
> 代码覆盖率是一个指标，用于描述特定测试套件在多大程度上对程序源代码做了测试。
>
> ——摘自http://en.wikipedia.org/wiki/Code_coverage

假设有一个端到端测试，覆盖了一部分简单代码，其代码覆盖率将很高，但并不能让你高枕无忧，因为很多其他部分未覆盖。

前面在代码库中引入了遗留代码——异常消息。这样做也许没错，只要它不是被依赖的行为，即没有依赖该异常消息：调试程序的程序员不依赖它，日志不依赖它，最终用户也不依赖它。不久的将来，程序中未被测试覆盖的部分可能退化，只要你能接受这样的风险即可。

也许，只要知道哪个代码行出现了什么样的异常就够了。有鉴于此，我们决定删除这条未被测试的异常消息：

```
public class TicTacToe {

    public void validatePosition(int x, int y) {
        if (isOutsideTheBoard(x)) {
            throw new RuntimeException("");
        }
        if (isOutsideTheBoard(y)) {
            throw new RuntimeException("");
}
    }

    private boolean isOutsideTheBoard
      (final int position) {
        return position < 1 || position > 3;
    }
}
```

1. 识别遗留代码的其他方式

你可能熟悉下面一些昭示着遗留应用程序的迹象：

- 补丁层补丁，俨然是作法自毙的科学怪人；
- 已知的bug；
- 修改代价高昂；
- 脆弱；

- ❑ 难以理解；
- ❑ 文档老旧、过期、一成不变，甚至根本没有文档；
- ❑ 霰弹式修改；
- ❑ 破窗效应。

这给应用程序维护团队带来如下影响：

- ❑ **放弃**：摆在软件负责者面前的是巨大的任务；
- ❑ **没人关心**：系统一旦出现受损的窗户，就更容易出现其他受损窗户。

遗留代码处理起来通常比其他软件更棘手，因此你可能想让最优秀的员工负责处理。然而，我们常常被最后期限弄得手忙脚乱，只想尽快将必不可少的功能开发出来，而对解决方案的质量不闻不问。

因此，为避免以如此糟糕的方式浪费优秀人才，我们希望非遗留应用程序完全相反，即具备如下特点：

- ❑ 易于修改；
- ❑ 可推广、可配置、可扩展；
- ❑ 易于部署；
- ❑ 健壮；
- ❑ 没有已知的缺陷或局限；
- ❑ 很容易讲解，也很容易了解；
- ❑ 大量测试套件；
- ❑ 自动验证；
- ❑ 可采取“微创手术”进行修改。

前面列出了遗留代码和非遗留代码的一些特征，好像很容易将某些特征变成其他特征，不是吗？停止霰弹式修改，转而采取“微创手术”，再添加一些细节，就大功告成了。果真如此吗？

可没有那么容易。好在有一些技巧和规则，我们可使用它们改善代码，让应用程序更类似于非遗留的。

2. 依赖不是注入的

这是遗留代码库最常见的一种坏味：对于不需要在隔离情况下进行测试的类，在需要时直接实例化协作者，导致使用协作者的类同时负责创建。

比如使用new运算符：

```
public class BirthdayGreetingService {

  private final MessageSender messageSender;
```

```
  public BirthdayGreetingService() {
    messageSender = new EmailMessageSender();
  }

  public void greet(final Employee employee) {
    messageSender.send(employee.getAddress(),
    "Greetings on your birthday");
  }
}
```

你无法对这个`BirthdayGreeting`服务进行单元测试。它依赖的`EmailMessageSender`是在构造函数中创建的，如果不修改代码库，就无法替换这个依赖（除非使用反射注入对象或在new运算符中替换对象）。

修改代码库是导致退化的温床，一定要三思而后行。要重构，就需要有测试（除非根本没法编写测试）。

> **遗留代码困境**
> 修改代码前，必须准备好测试；而要准备好测试，通常要修改代码。

3. 遗留代码修改算法

必须对遗留代码进行修改时，可使用如下算法：

- 确定修改点；
- 找出测试点；
- 消除依赖；
- 编写测试；
- 修改并重构。

4. 应用遗留代码修改算法

为应用这种算法，通常先编写一组测试，并在重构期间确保这些测试始终能通过。这不同于正常的TDD周期，因为重构不能引入任何新功能，即不应编写任何新规范。

为更好地解释这种算法，假设我们被要求做如下修改：

为更轻松问候员工，我想给他们发推特消息而不是电子邮件。

(1) 确定修改点

当前，这个系统只能发送电子邮件，因此必须修改。在哪里修改呢？经过简单的调查可发现，

问候方式是在BirthdayGreetingService类的构造函数中决定的，这个构造函数采用了策略模式（https://en.wikipedia.org/?title=Strategy_pattern），如下述代码片段所示：

```
public class BirthdayGreetingService {

  public BirthdayGreetingService() {
    messageSender = new EmailMessageSender();
  }
  [...]
}
```

(2) 找出测试点

BirthdayGreetingService类没有任何注入的协作者可为其添加额外的功能，因此只能在这个服务类外部对其进行测试。一种办法是修改EmailMessageSender类，将其实现替换为模拟或伪造实现，但这是拿这个类的实现冒险。

另一种选择是为这项功能编写一个端到端测试：

```
public class EndToEndTest {

  @Test
  public void email_an_employee() {
    final StringBuilder systemOutput =
       injectSystemOutput();
    final Employee john = new Employee(
       new Email("john@example.com"));

    new BirthdayGreetingService().greet(john);

    assertThat(systemOutput.toString(),
      equalTo("Sent email to "
        + "'john@example.com' with "
+ "the body 'Greetings on your "
        + "birthday'\n"));
  }

  //这些代码来自GMaur's LegacyUtils (https://github.com/GMaur/legacyutils)
  //获得了它们的许可：
  private StringBuilder injectSystemOutput() {
    final StringBuilder stringBuilder =
      new StringBuilder();
    final PrintStream outputPrintStream =
      new PrintStream(
new OutputStream() {
        @Override
        public void write(final int b)
          throws IOException {
          stringBuilder.append((char) b);
        }
      });
```

8

```
    System.setOut(outputPrintStream);
    return stringBuilder;
  }
}
```

我们在获得许可的情况下从https://github.com/GMaur/legacyutils借鉴了这些代码。这个库旨在帮助你捕捉系统输出（System.out）。

这个文件的文件名不像TicTacToeSpec等那样以Spec结尾。这个测试旨在确保相应功能保持不变，其所属文件名为EndToEndTest，因为我们力图覆盖尽可能多的功能。

(3) 消除依赖

创建旨在确保预期行为不变的测试后，下面解除BirthdayGreetingService和EmailMessageSender之间的硬编码式依赖。为此，我们将使用一种名为“**提出并重写调用**”的技术，这种技术是Michaels Feathers在其著作中首次提出的：

```
public class BirthdayGreetingService {

  public BirthdayGreetingService() {
    messageSender = getMessageSender();
  }

  private MessageSender getMessageSender() {
    return new EmailMessageSender();
  }

[...]
```

再次运行测试，我们编写的唯一一个测试通过了。为让这个方法可重写，需要将其改为受保护或公有的：

```
public class BirthdayGreetingService {

  protected MessageSender getMessageSender() {
    return new EmailMessageSender();
  }

[...]
```

在测试文件夹中创建一个伪造对象。使用代码引入伪造对象是一种模式：创建一个对象，它可用于替代既有对象，且其行为是可控的。这样就能通过注入一些自定义的伪造对象满足我们的需求。有关这个模式的更详细信息，请参阅http://xunitpatterns.com/。

在这里，我们应创建一个伪造的服务，它扩展原来的服务。下一步是重写复杂的方法，以绕开与测试无关的代码：

```
public class FakeBirthdayGreetingService
extends BirthdayGreetingService {
```

```
  @Override
  protected MessageSender getMessageSender() {
    return new EmailMessageSender();
  }
}
```

现在，我们可以使用这个伪造对象，而不使用`BirthdayGreetingService`类：

```
public class EndToEndTest {

  @Test
  public void email_an_employee() {
    final StringBuilder systemOutput =
      injectSystemOutput();
    final Employee john = new Employee(
       new Email("john@example.com"));

    new FakeBirthdayGreetingService().greet(john);

    assertThat(systemOutput.toString(),
      equalTo("Sent email to "
        + "'john@example.com' with "
        + "the body 'Greetings on "
        + "your birthday'\n"));
  }
```

这个测试也通过了。

现在，可使用Feathers在其论文（http://www.objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf）中阐述的另一个依赖解除技巧——参数化构造函数。产品代码可能如下所示：

```
public class BirthdayGreetingService {

  public BirthdayGreetingService(final MessageSender
    messageSender) {
    this.messageSender = messageSender;
  }
  [...]
}
```


上述实现对应的测试代码如下所示：

```
public class EndToEndTest {

  @Test
  public void email_an_employee() {
    final StringBuilder systemOutput =
      injectSystemOutput();
    final Employee john = new Employee(
      new Email("john@example.com"));
```

```
    new BirthdayGreetingService(new
        EmailMessageSender()).greet(john);

    assertThat(systemOutput.toString(),
      equalTo("Sent email to "
        + "'john@example.com' with "
        + "the body 'Greetings on "
        + "your birthday'\n"));
  }
  [...]
```

现在不再需要FakeBirthday，可以将其删除。

(4) 编写测试

我们保留前面的端到端测试，同时创建一个交互测试验证BirthdayGreetingService和MessageSender之间的集成情况：

```
@Test
public void the_service_should_ask_the_messageSender() {
  final Email address =
    new Email("john@example.com");
  final Employee john = new Employee(address);
  final MessageSender messageSender =
    mock(MessageSender.class);

  new BirthdayGreetingService(messageSender)
    .greet(john);

  verify(messageSender).send(address,
      "Greetings on your birthday");
}
```

现在可以编写新的TweetMessageSender类，完成遗留代码修改算法的最后一步。

8.2　编码套路

程序员要提高技能，只能通过练习，别无他法。使用不同技术创建不同类型的程序，通常让程序员对软件开发有新的洞见。编码套路是一种秉承这种理念的练习，定义了为达成某种目标而必须实现的需求或功能。

程序员被要求实现一种可能的解决方案，再将其与其他解决方案进行比较，力图找出最佳解决方案。这种练习的重点不是以最快速度实现解决方案，而是对设计解决方案期间做出的决策进行讨论。大多数情况下，在编码套路中创建的所有程序最终都会被丢弃。

本章的编码套路针对的是一个遗留系统。这个程序足够简单，让你在本章就能处理完毕；同时又足够复杂，能够给你出些难题。

8.2.1 遗留代码处理套路

你接受了一项任务——对一个已部署到生产环境的系统进行修改。这个系统是一款图书馆软件——名为Alexandria的项目。

这个项目当前没有文档，原来的维护人员也联系不上，无法与之讨论。因此，如果你接受这项任务，就得完全靠自己，因为没人可以指望。

8.2.2 描述

我们想方设法找到了最初编写这个项目时制定的如下规范片段：

- 软件Alexandria必须能够存储图书并将其借给能够归还的用户。用户还需能够在这个系统根据作者、书名、状态和ID搜索图书。
- 对于归还图书的时间没有限制。
- 还能将图书下架，因为出于商业上的考虑，这很重要。
- 这个软件不应接纳新用户。
- 应随时将服务时间告知用户。

8.2.3 技术说明

Alexandria是一个使用Java编写的后端项目，使用REST API向前端提供信息。为简化这个编码套路，使用测试替身（伪造对象）、以内存对象的方式实现持久化。有关伪造对象的更详细信息，请参阅http://xunitpatterns.com/Fake%20Object.html。

https://bitbucket.org/vfarcic/tdd-chapter-08/commits/branch/legacy-code提供了这个项目的现有代码。

8.2.4 添加新功能

如果无需添加新功能，这些遗留代码可能不会给程序员带来困扰。这个代码库的状态虽不理想，但生产系统运行正常，也没有带来什么麻烦。

现在问题来了：**产品所有者（PO）**要添加一项新功能。

例如，给定一本图书，图书管理员想知道其完整的出借历史，以便确定哪些图书更受用户欢迎。

8.2.5 黑盒测试还是尖峰冲击测试

Alexandria项目没有文档，也无法向以前的维护人员咨询，这加大了黑盒测试的难度。有鉴于此，我们决定通过调查研究更深入地了解这个软件，再做些尖峰冲击获悉系统内部情况。

后面将根据由此获得的信息实现新功能。

> 黑盒测试这种软件测试方法在不探查应用程序内部结构和工作原理的情况下，对其功能进行检查。这种测试方法适用于各个层级的软件测试：单元测试、集成测试、系统测试和验收测试。较高层级的测试大多乃至全部都是黑盒测试，单元测试也可能大多是黑盒测试。
>
> ——摘自http://en.wikipedia.org/wiki/Black-box_testing
>
> 有关黑盒测试的更详细信息，请参阅http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf。

8.2.6　初步调查

知道需要添加的新功能后，开始调查项目Alexandria：

- 15个文件；
- 基于maven（`pom.xml`）；
- 没有测试。

首先，我们要确认这个项目根本没有测试过；项目中没有test文件夹印证了这点：

```
$ find src/test
find: src/test: No such file or directory
```

下面列出Java部分的文件夹：

```
$ cd src/main/java/com/packtpublishing/tddjava/ch08/alexandria/
$ find .
.
./Book.java
./Books.java
./BooksEndpoint.java
./BooksRepository.java
./CustomExceptionMapper.java
./MyApplication.java
./States.java
./User.java
./UserRepository.java
./Users.java
```

其他部分的文件夹如下所示：

```
$ cd src/main
$ find resources webapp
resources
resources/applicationContext.xml
webapp
```

```
webapp/WEB-INF
webapp/WEB-INF/web.xml
```

这好像是个Web项目（文件`web.xml`表明了这一点），使用的是Spring（`applicationContext.xml`表明了这一点）。下面列出了pom.xml包含的部分依赖：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>4.1.4.RELEASE</version>
</dependency>
```

其中有Spring，这是个好兆头，因为它可帮助注入依赖，但快速调查表明并没有真正使用这个上下文。莫非这是以前使用的？

在文件`web.xml`中，我们发现了如下片段：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <module-name>alexandria</module-name>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>SpringApplication</servlet-name>
        <servlet-class>
org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.packtpublishing.tddjava.ch08.alexandria.MyApplication
                </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

这个文件表明：

- 将加载`applicationContext.xml`中的上下文；
- 将在一个servlet中执行应用程序文件`com.packtpublishing.tddjava.ch08. Alexandria.MyApplication`。

文件`MyApplication`的内容如下所示：

```
public class MyApplication extends ResourceConfig {
```

```
    public MyApplication() {
        register(RequestContextFilter.class);
        register(BooksEndpoint.class);
        register(JacksonJaxbJsonProvider.class);
        register(CustomExceptionMapper.class);
    }
}
```

它配置必要的类，以便执行端点`BooksEndpoint`（片段）：

```
@Path("books")
@Component
public class BooksEndpoint {

    private BooksRepository books = new BooksRepository();

    private UserRepository users = new UserRepository();
```

这个代码片段中，我们发现了遗留代码的特征之一：两个依赖（`books`和`users`）都是在端点内创建的，而不是注入的。这导致单元测试更为困难。

我们可记录重构期间要使用的元素，以便后面编写将依赖注入`BooksEndpoint`的代码。

1. 如何确定可重构的地方

编程范式（如函数式、命令式和面向对象）和风格（如简洁、详尽、极简和自恋）很多，可重构的地方因人而异。

确定可重构的地方时，还有一种与主观相反的方式：客观的方式。有研究论文探讨了这些方式，如http://www.bth.se/fou/cuppsats.nsf/all/2e48c5bc1c234d0ec1257c77003ac842/$file/BTH2014SIVERLAND.pdf。

2. 引入新功能

对代码有更深入的了解后，看起来最重要的功能更改是，将当前使用的单个状态（即以下代码片段）：

```
@XmlRootElement
public class Book {

    private final String title;
    private final String author;
    private int status; // 这是一个属性
    private int id;
```

替换为状态集合（即下面的代码片段）：

```
@XmlRootElement
public class Book {
```

```
    private int[] statuses;
    // ...
```

这看起来管用（将对字段的访问改为对数组的访问后），但也提出了另一个功能需求。

软件Alexandria必须能够存储图书并将其借给能够归还的用户。用户还能在这个系统中根据作者、书名、状态和ID搜索图书。

产品所有者（PO）确认现在根据状态搜索图书的方式变了：现在还允许搜索以前的状态。

这导致需要做的修改越来越多。每当觉得该将遗留代码删除时，我们就开始应用遗留代码修改算法。

我们还发现了坏味“依恋情结”和“基本类型偏执”：使用int变量存储状态（基本类型偏执），并修改另一个对象的状态（依恋情结）。我们将这个可重构的地方加入待办事项清单。

- ❑ 将依赖注入BooksEndpoint；
- ❑ 将单个状态改为多个状态；
- ❑ 消除状态的“基本类型偏执”坏味（可选）。

8.2.7　应用遗留代码修改算法

在这里，整个后端是独立工作的，它使用的是内存持久化；即便数据被保存到数据库，也可使用同样的算法，但需要编写一些额外的代码，用于在测试之间清理和填充数据库。

我们将使用DbUnit，有关这个测试框架的更详细信息，请参阅http://dbunit.sourceforge.net/。

1. 编写端到端测试用例

为确保行为在重构期间不变，我们决定首先编写端到端测试。对于其他包含前端的应用程序，可使用更高级的工具（如Selenium/Selenide）完成这项工作。

在这里，由于前端不需要重构，因此可使用较低级的工具。我们选择编写HTTP请求进行端到端测试。

这些请求应是自动的、可测试的，因此应遵循所有自动测试规则。鉴于要在编写这些测试的同时探索应用程序的实际行为，我们决定使用工具Postman（这个工具可在如下网址找到：https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm，其官网为https://www.getpostman.com/）编写一个尖峰冲击。也可使用工具curl（http://curl.haxx.se/）编写尖峰冲击。

> **curl是什么？**
>
> curl是一个使用URL语法传输数据的命令行工具和库，支持...HTTP、HTTPS、... HTTP POST、HTTP PUT...。
>
> **curl有何用途呢？**
>
> 可在命令行或脚本中使用curl传输数据。
>
> ——摘自http://curl.haxx.se/

为此，我们决定使用如下命令，在本地执行这个遗留软件：

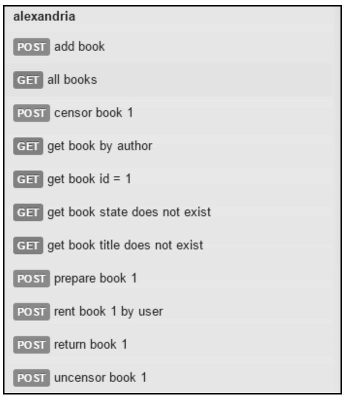
```
mvn clean jetty:run
```

这将启动一个处理请求的本地jetty服务器。这样做的最大好处是，部署是自动完成的，无需将一切打包并手动将其部署到应用程序服务器（如JBoss AS、GlassFish、Geronimo或TomEE）。这可极大提高修改并查看效果的速度，从而缩短反馈时间。在本章后面，我们将在Java代码中以编程方式启动这个服务器。

先确定有哪些功能。在本章前面，我们发现`BooksEndpoint`类包含Web服务端点的定义，从这里着手确定有哪些功能是不错的选择。发现的功能如下。

(1) 添加新书。
(2) 列出所有图书。
(3) 根据ID、作者、书名和状态搜索图书。
(4) 为出租图书做准备。
(5) 出租图书。
(6) 将图书下架。
(7) 将图书重新上架。

我们手动启动服务器，并开始编写请求。



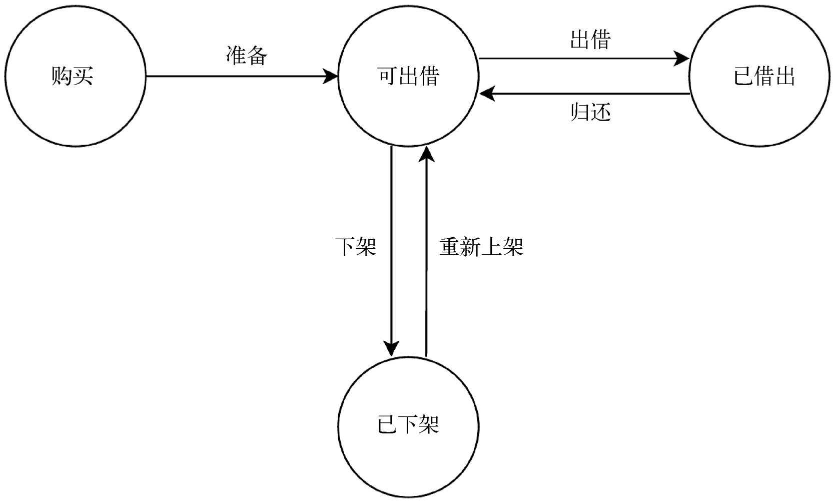
这些测试看起来足以达到尖峰冲击的目的。我们注意到每个响应都包含时间戳，这让自动化更为棘手。



测试要更有价值，必须自动化并面面俱到。但当前不是这样的，因此我们将它们视为尖峰冲击。本章后面将自动化这些测试。

> 我们执行的每个测试都不是自动化的，但在这里，使用Postman界面编写测试的速度比自动化测试快得多。另外，与实际使用这个产品相比，获得的体验也更有代表性。测试客户端可能给产品带来问题，导致返回的结果不可信。
>
> 此处，我们发现使用Postman测试是更合算的投资，因为使用完毕将丢弃这些测试。这些测试以极快的速度提供有关API的反馈和结果。我们还使用Postman创建REST API原型，因为它提供的工具既有效又管用。
>
> 一般而言，根据是否需要将测试留到以后使用而选择不同工具；需要考虑的因素还包括测试的执行频率以及执行环境。

执行前述请求后，我们发现了这个应用程序的一些状态，如下图所示。



通过尖峰冲击对应用程序有所了解后，该自动化测试了。毕竟，如果不自动化测试，就不能信心满满地去重构。

2. 自动化测试用例

我们将以编程方式启动服务器。为此，使用Grizzly（https://grizzly.java.net/），它让我们能够使用来自Jersey ResourceConfig（FQCN: org.glassfish.jersey. server.ResourceConfig）的配置以启动服务器，如测试类BooksEndpointTest所示（这只是一个片段，完整的源代码可在https://bitbucket.org/vfarcic/tdd-chapter-08/commits/branch/refactor/inject-dependencies找到）：

```
public class BooksEndpointTest {
    public static final URI FULL_PATH =
      URI.create("http://localhost:8080/alexandria");
    private HttpServer server;

    @Before
    public void setUp() throws IOException {
        ResourceConfig resourceConfig =
          new MyApplication();
        server = GrizzlyHttpServerFactory
          .createHttpServer(FULL_PATH, resourceConfig);
        server.start();
    }

    @After
    public void tearDown(){
        server.shutdownNow();
    }
```

这些代码启动一个本地服务器，其地址为http://localhost:8080/alexandria。这个服务器仅在

很短的时间内（测试运行时）可用，因此如果你需要手动访问，请在需要暂停执行时调用下面的方法：

```
public void pauseTheServer() throws Exception {
    System.in.read();
}
```

如果要停止这个服务器，可停止执行，也可在控制台按回车键。

现在我们能够以编程方式启动这个服务器，请（使用前面的方法）将其暂停，并再次执行前面的尖峰冲击。结果没变，说明重构成功。

下面给系统添加第一个自动化测试（代码可在https://bitbucket.org/vfarcic/tdd-chapter-08/commits/branch/refactor/inject-dependencies找到）：

```
public class BooksEndpointTest {

    public static final String AUTHOR_BOOK_1 =
      "Viktor Farcic and Alex Garcia";
    public static final String TITLE_BOOK_1 =
      "TDD in Java";
    private final Map<String, String> TDD_IN_JAVA;

    public BooksEndpointTest() {
        TDD_IN_JAVA = getBookProperties(TITLE_BOOK_1,
AUTHOR_BOOK_1);
    }

    private Map<String, String> getBookProperties
      (String title, String author) {
        Map<String, String> bookProperties =
          new HashMap<>();
        bookProperties.put("title", title);
        bookProperties.put("author", author);
        return bookProperties;
    }

    @Test
    public void add_one_book() throws IOException {
        final Response books1 = addBook(TDD_IN_JAVA);
        assertBooksSize(books1, is("1"));
    }

     private void assertBooksSize(Response response,
       Matcher<String> matcher) {
        response.then().body(matcher);
    }

    private Response addBook
      (Map<String, ?> bookProperties) {
        return RestAssured
```

8

```
                .given().log().path()
                .contentType(ContentType.URLENC)
                .parameters(bookProperties)
                .post("books");
    }
```

为进行测试，我们使用了一个名为RestAssured的库（https://code.google.com/p/rest-assured/），它让我们能够更轻松地测试REST和JSON。

为完成这个自动化测试套件，我们创建如下测试：

(1) `add_one_book()`
(2) `add_a_second_book()`
(3) `get_book_details_by_id()`
(4) `get_several_books_in_a_row()`
(5) `censor_a_book()`
(6) `cannot_retrieve_a_censored_book()`

可在https://bitbucket.org/vfarcic/tdd-chapter-08/commits/branch/refactor/inject-dependencies找到它们的代码。

有了确保不会导致退化的测试套件后，下面查看待办事项清单：

- 将依赖注入BooksEndpoint；
- 将单个状态改为多个状态；
- 消除状态的“基本类型偏执”坏味（可选）。

先处理依赖注入。

3. 注入依赖BookRepository

依赖`BookRepository`是在`BooksEndpoint`中创建的，如下代码片段所示：

```
@Path("books")
@Component
public class BooksEndpoint {

    private BooksRepository books =
      new BooksRepository();
[...]
```

8.2.8　提取并重写调用

我们将使用前面介绍的重构方法“提取并重写调用”。为此，创建一个失败的规范，如下所示：

```
@Test
public void add_one_book() throws IOException {
    addBook(TDD_IN_JAVA);

    Book tddInJava = new Book(TITLE_BOOK_1,
      AUTHOR_BOOK_1,
       States.fromValue(1));

    verify(booksRepository).add(tddInJava);
}
```

为让这个处于红灯状态的规范（失败的规范）通过，首先提出创建依赖的代码，并将其放在BookRepository类的一个受保护的方法中：

```
@Path("books")
@Component
public class BooksEndpoint {

    private BooksRepository books =
      getBooksRepository();

    [...]

     protected BooksRepository
       getBooksRepository() {
        return new BooksRepository();
    }

    [...]
```

复制启动器MyApplication的代码：

```
public class TestApplication
    extends ResourceConfig {

    public TestApplication
      (BooksEndpoint booksEndpoint) {
        register(booksEndpoint);
        register(RequestContextFilter.class);
        register(JacksonJaxbJsonProvider.class);
        register(CustomExceptionMapper.class);
    }

    public TestApplication() {
        this(new BooksEndpoint(
          new BooksRepository()));
    }
}
```

这让我们能够注入任何BooksEndpoint。这个示例中，我们将在BooksEndpoint-InteractionTest中重写依赖获取方法。这样就能检查是否调用了必要的方法，如Books-EndpointInteractionTest中的下述代码片段所示：

8

```
@Test
public void add_one_book() throws IOException {
    addBook(TDD_IN_JAVA);
    verify(booksRepository)
      .add(new Book(TITLE_BOOK_1,
          AUTHOR_BOOK_1, 1));
}
```

运行测试，一路绿灯。虽然规范都通过了，但我们引入了一个仅用于测试的设计片段；可是产品代码不会执行新增的启动器TestApplication，而依然执行旧启动器MyApplication。要解决这个问题，我们并需将这两个启动器合而为一。为此，可使用重构方法“参数化构造函数，这在Roy Osherove的著作《单元测试的艺术》(http://www.ituring.com.cn/book/1336) 中也有介绍。

- **参数化构造函数**

我们可将接受依赖BooksEndpoint的启动器合而为一：如果没有指定依赖，就使用实际的BooksRepository实例注册依赖，否则注册收到的依赖：

```
public class MyApplication
      extends ResourceConfig {

    public MyApplication() {
        this(new BooksEndpoint(
          new BooksRepository()));
    }

    public MyApplication
      (BooksEndpoint booksEndpoint) {
        register(booksEndpoint);
        register(RequestContextFilter.class);
        register(JacksonJaxbJsonProvider.class);
        register(CustomExceptionMapper.class);
    }
}
```

在这里，我们使用“构造函数串接”避免构造函数包含重复代码。

执行这种重构后，BooksEndpointInteractionTest类如下所示 (这种最终版本):

```
public class BooksEndpointInteractionTest {

    public static final URI FULL_PATH = URI.
        create("http://localhost:8080/alexandria");
    private HttpServer server;
    private BooksRepository booksRepository;

    @Before
    public void setUp() throws IOException {
        booksRepository = mock(BooksRepository.class);
        BooksEndpoint booksEndpoint =
          new BooksEndpoint(booksRepository);
```

```
        ResourceConfig resourceConfig =
          new MyApplication(booksEndpoint);
        server = GrizzlyHttpServerFactory
           .createHttpServer(FULL_PATH, resourceConfig);
        server.start();
    }
```

第一个测试通过，因此可将依赖注入任务标记为“已完成”。

代办事项清单如下：

- ❑ 将依赖注入BooksEndpoint；
- ❑ 将单个状态改为多个状态；
- ❑ 消除状态的“基本类型偏执”坏味（可选）。

添加新功能

有了必要的测试环境后，即可添加新功能。

给定一本图书，图书管理员想知道其完整的出借历史，以确定哪些图书更受欢迎。

首先编写一个不能通过的规范：

```
public class BooksSpec {

    @Test
    public void should_search_for_any_past_state() {
        Book book1 = new Book("title", "author",
          States.AVAILABLE);
        book1.censor();

        Books books = new Books();
        books.add(book1);

        String available =
          String.valueOf(States.AVAILABLE);
        assertThat(
          books.filterByState(available).isEmpty(),
          is(false));
    }
}
```

运行所有测试，发现最后一个失败。

实现根据各种状态进行搜索的功能：

```
public class Book {

    private ArrayList<Integer> status;

    public Book(String title, String author, int status) {
```

```
        this.title = title;
        this.author = author;
        this.status = new ArrayList<>();
        this.status.add(status);
    }

    public int getStatus() {
        return status.get(status.size()-1);
    }

     public void rent() {
        status.add(States.RENTED);
    }
    [...]

    public List<Integer> anyState() {
        return status;
    }
    [...]
```

这个代码片段中，我们没有列出不相关的部分：未修改的或以类似方式修改实现的方法（如rent）。

```
public class Books {
    public Books filterByState(String state) {
        Integer expectedState = Integer.valueOf(state);
        return new Books(
            new ConcurrentLinkedQueue<>(
                books.stream()
                  .filter(x
                -> x.anyState()
                  .contains(expectedState))
                  .collect(toList())));
    }
    [...]
```

外部方法（尤其是序列化为JSON的方法）不受影响，因为方法getStatus的返回类型依然是int。

运行所有测试，发现一路绿灯。

代办事项清单如下：

- 将依赖注入BooksEndpoint；
- 将单个状态改为多个状态；
- 消除状态的“基本类型偏执”坏味（可选）。

8.2.9　消除状态的“基本类型偏执”坏味

我们决定，也要完成待办事项清单中的可选项。

代办事项清单如下：

- 将依赖注入BooksEndpoint；
- 将单个状态改为多个状态；
- 消除状态的“基本类型偏执”坏味（可选）。

> “基本类型偏执”坏味指的是使用基本数据类型表示域概念。例如，使用字符串表示消息，使用整数表示金额，使用结构体/字典/散列表示对象。
>
> ——摘自http://c2.com/cgi/wiki?PrimitiveObsession

这是一个重构步骤（即不会在系统中引入新行为），因此不需要编写新规范。我们将力保始终处于绿灯状态，或不处于绿灯状态的时间很短。

当前，`States`是一个包含常量的`java`类，如下所示：

```
public class States {
    public static final int BOUGHT = 1;
    public static final int RENTED = 2;
    public static final int AVAILABLE = 3;
    public static final int CENSORED = 4;
}
```

将其转换为枚举：

```
enum States {
    BOUGHT (1),
    RENTED (2),
    AVAILABLE (3),
    CENSORED (4);

    private final int value;

    private States(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public static States fromValue(int value) {
        for (States states : values()) {
            if(states.getValue() == value) {
                return states;
            }
        }
        throw new IllegalArgumentException(
          "Value '" + value
    + "' could not be found in States");
    }
}
```

再将测试相应修改如下：

```
public class BooksEndpointInteractionTest {
    @Test
    public void add_one_book() throws IOException {
        addBook(TDD_IN_JAVA);
        verify(booksRepository).add(
            new Book(TITLE_BOOK_1, AUTHOR_BOOK_1,
              States.BOUGHT));
    }
    [...]
public class BooksTest {

    @Test
    public void should_search_for_any_past_state() {
        Book book1 = new Book("title", "author",
          States.AVAILABLE);
        book1.censor();

        Books books = new Books();
        books.add(book1);

        assertThat(books.filterByState(
            String.valueOf(
              States.AVAILABLE.getValue()))
            .isEmpty(), is(false));
    }
    [...]
```

接下来，修改产品代码，如下代码片段所示：

```
@XmlRootElement
public class Books {
      public Books filterByState(String state) {
        State expected =
          States.fromValue(Integer.valueOf(state));
        return new Books(
            new ConcurrentLinkedQueue<>(
                books.stream()
                  .filter(x -> x.anyState()
                    .contains(expected))
                  .collect(toList())));
    }
    [...]
```

还有如下代码片段：

```
@XmlRootElement
public class Book {

    private final String title;
    private final String author;
    @XmlTransient
```

```
    private ArrayList<States> status;
    private int id;

    public Book
      (String title, String author, States status) {
        this.title = title;
        this.author = author;
        this.status = new ArrayList<>();
        this.status.add(status);
    }

    public States getStatus() {
        return status.get(status.size() - 1);
    }

    @XmlElement(name = "status")
    public int getStatusAsInteger(){
        return getStatus().getValue();
    }

    public List<States> anyState() {
        return status;
    }
    [...]
```

在这里，序列化是使用注解实现的：

```
@XmlElement(name = "status")
```

将方法的结果转换为status字段。

另外，对于字段status（现在为ArrayList<States>，使用@XmlTransient进行标记，因为它不会序列化为JSON。

执行所有测试，它们都通过了，因此现在可以将待办事项清单中的可选项也勾掉。

待办事项清单如下：

- ❑ 将依赖注入BooksEndpoint；
- ❑ 将单个状态改为多个状态；
- ❑ 消除状态的“基本类型偏执”坏味（可选）。

8.3 小结

你知道，接手遗留代码库可能是一项令人怯步的任务。

本章前面说过，遗留代码是不带测试的代码。因此要处理遗留代码，首先需要创建测试，确保遗留代码的功能在处理过程中保持不变。

遗憾的是，创建测试并非总是那么容易。遗留代码通常紧密耦合，还存在昭示着设计不善（至少以前没关心过代码质量）的症状。但不用担心，你可逐步执行一些繁琐的步骤，详情请参阅http://martinfowler.com/bliki/ParallelChange.html。另外，众所周知，软件开发就是一个学习过程，能够正确运行的代码不过是这个过程的副产品。因此，最重要的是更深入地了解代码库，以便能够安全修改。更详细的信息请参阅http://www.slideshare.net/ziobrando/model-storming。

最后，强烈建议你阅读Michael Feathers的著作《修改代码的艺术》。其中介绍了大量处理遗留代码库的技巧，对理解整个过程大有裨益。

第 9 章

功能开关——将未完成的功能部署到生成环境

“不要受环境控制，去改变环境。”

——成龙

至此，你知道TDD让开发过程更容易，还可缩短编写高质量代码所需的时间。它还能带来另一个好处：由于代码经过了测试，其正确性得到广泛证明，因此我们可进一步认为，所有测试都通过后，便可将代码部署到生产环境。

有一些基于这种理念的软件生命周期方法。本章将介绍一些极限编程实践，如**持续集成**、**持续交付**和**持续部署**。

本章涵盖如下主题：

- 持续集成、持续交付和持续部署；
- 在生产环境中测试应用程序；
- 功能开关。

9.1　持续集成、持续交付和持续部署

测试驱动开发与**持续集成**（**CI**）、**持续交付**和**持续部署**（**CD**）相辅相成。持续集成、持续交付和持续部署虽然有些差别，但它们的目标类似，即都力图不断验证代码可否部署到生产环境。从这种意义上说，它们与TDD很像，都提倡采用极短的开发周期持续验证当前编写的代码，旨在确保应用程序始终处于可部署到生产环境的状态。

由于篇幅有限，本书无法详细介绍这些技术。实际上，有关这个主题可编写一部专著。这里只介绍这三种技术的差别。持续集成意味着始终将代码与系统的其他部分集成起来，让问题快速浮出水面。发现问题后，将优先修复导致问题的原因，推后新的开发工作。你可能注意到，这个

定义与TDD的工作原理类似，主要差别在于TDD的关注点并非与系统其他部分的集成，此外的方面都相同。TDD和持续集成都力图快速发现问题，并将修复问题放在最优先的位置——其他事情都得靠边站。持续集成并未将整个流水线都自动化——将代码部署到生产环境前，需要执行额外的手动验证。

持续交付很像持续集成，但比后者更进一步，将整个流水线都自动化（部署到生产环境除外）。每次将代码提交到仓库并通过所有验证后，便可部署到生产环境。然而，部署决策是人工完成的，需要有人选择要部署到生产环境的构建。选择是基于策略或功能的，即要将哪个构建提供给用户以及何时提供——虽然所有构建都可部署到生产环境。

> “持续交付是一种软件开发准则，通过采用合适的开发方式确保软件在任何时候都可发布到生产环境。”
>
> ——Martin Fowler

最后，如果有关部署内容的决策也是自动做出的，持续交付就变成持续部署。这种情况下，每次提交代码后，如果通过所有验证就将部署到生产环境，无一例外。

要持续将代码交付到生产环境，必须满足如下条件：要么没有分支，要么分支从被创建到被集成到主干（mainline）的时间很短（不超过一天，最好只有几小时）；否则就不能持续验证代码。

提交代码前必须创建验证，这是将这些技术与TDD联系起来的纽带。如果不预先创建这些验证，提交到仓库的代码将没有配套测试，整个过程将以失败告终。如果没有测试，我们就没法对开发的代码充满信心；如果没有TDD，就没有与实现代码配套的测试。还有另一种选择——推迟提交，即等到测试创建后再提交到仓库。但这样做根本谈不上“持续”：测试编写好之前，代码一直存储在开发人员的计算机中，根本无法持续对整个系统进行验证。

总之，持续集成、持续交付和持续部署依赖于与实现代码配套的测试，即依赖于TDD，还要求不使用分支或确保分支的存活时间极短（被频繁地合并到主干）。问题是，有些功能并不能在那么短的时间内开发出来。不管功能多小，有可能都需要几天的开发时间；在此期间，我们不能将其提交到仓库，否则它们将被交付到生产环境——用户可不想看到不完整的功能。例如，交付不完整的登录功能毫无意义；如果用户看到一个登录页面，其中包含用户名和密码文本框，还有一个登录按钮，而单击这个按钮并不能存储输入的信息并生成身份验证cookie，那么即便在最好的情况下，这也只会让用户感到迷惑。有些功能在没有其他功能的情况下不管用；继续前面的例子，即便登录功能开发好了，如果没有注册功能，它也毫无意义。

想想玩拼图的情况吧。你需要对最终拼出来的图形有大致认识，但每次都专注于一块拼图。你选择自己认为最容易确定位置的拼图，并将其与周边的拼图合并。仅当所有拼图都各就各位，

完整的图形才能出现，至此大功告成。

TDD亦如此。我们通过每次专注于一个小单元来开发代码。随着工作的进行，整个软件初具雏形，各个单元相互配合，最终完全集成。整个过程结束前，即便所有测试都通过，我们看到的是绿灯，也不能将代码交付给最终用户。

为解决这些问题，同时又让TDD和CI/CD的效果不打折扣，最简单的办法就是使用功能开关。

9.2 功能开关

功能开关（Feature Toggle）也叫功能切换（Feature Flipping）或功能标志（Feature Flag）。虽然叫法不同，但它们都基于这样一种机制，即让你能够开启和关闭应用程序的功能。所有代码都被合并到一个分支，而你必须处理未全部完成（或集成）的代码时，这很有用。使用这种技巧可隐藏未完成的功能，让用户无法访问。

这种功能的特征使它还有其他用途：在功能存在问题时充当断路器，让应用程序平稳退化；关闭次要功能，将硬件资源留给核心业务。有些情况下，功能开关还能走得更远。例如，根据用户的地理位置或扮演的角色决定是否启用特定功能。另一个用途是仅对测试者启用新功能，这样最终用户根本不知道这些新功能的存在，同时让测试者能够在生产服务器上验证。

使用功能开关时，需要牢记一些要点：

- 仅当功能已部署并确定管用后才使用开关。否则代码可能充斥着if/else语句，这些语句包含不再使用的旧开关。
- 不要花过多时间测试开关。大多数情况下，只需确定新功能的入口不可见即可，这种入口可能是到新功能的链接。
- 不要滥用开关。不要在不需要的情况下使用开关，例如，你可能正在开发一个新屏幕，这个屏幕可通过主页中的一个链接进行访问。如果这个链接位于主页末尾，可能就没有必要使用开关对其进行隐藏。

用于处理应用程序功能的优秀框架和库有很多，下面是其中的两个：

- Togglz（http://www.togglz.org/）；
- FF4J（http://ff4j.org/）。

9

这些库提供了复杂的功能管理方式，根据角色或规则决定是否开启功能。你通常不需要这样复杂的功能管理方式，但这让我们能够在生产环境中测试新功能，同时不对所有用户都开启。然而，自己动手实现基本的功能开关解决方案很容易，我们将通过一个示例证明。

9.3　功能开关示例

下面来看演示程序。我们将创建一个简单的小型**REST**服务，它根据用户的要求计算斐波那契数列中的第*N*个数。我们将使用一个文件记录被启用/禁用的功能。出于简化考虑，使用**spring-boot**框架和模版引擎**Thymeleaf**，这个模版引擎包含在依赖spring-boot中。有关spring-boot及相关项目的更详细信息，请参阅http://projects.spring.io/spring-boot/；有关模版引擎Thymeleaf的更详细信息，请参阅http://www.thymeleaf.org/。

文件`build.gradle`如下所示：

```
apply plugin: 'java'
apply plugin: 'application'

sourceCompatibility = 1.8
version = '1.0'
mainClassName = "com.packtpublishing.tddjava.ch09.Application"

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile group: 'org.springframework.boot',
            name: 'spring-boot-starter-thymeleaf',
            version: '1.2.4.RELEASE'

    testCompile group: 'junit',
    name: 'junit',
    version: '4.12'
}
```

注意，其中包含插件`application`，因为我们要使用Gradle命令run运行这个应用程序。这个应用程序的主类如下所示：

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

我们将创建属性文件。为此，我们将使用YAML格式，因为它简洁且很容易理解。请在文件夹`src/main/resources`中添加文件`application.yml`，并在其中添加如下内容：

```
features:
    fibonacci:
        restEnabled: false
```

Spring提供了一种自动加载这种属性文件的方式。当前只有两个约束条件：文件的名称必须

为application.yml；文件必须包含在应用程序的类路径中。

下面是功能配置文件的实现：

```
@Configuration
@EnableConfigurationProperties
@ConfigurationProperties(prefix = "features.fibonacci")
public class FibonacciFeatureConfig {
    private boolean restEnabled;

    public boolean isRestEnabled() {
        return restEnabled;
    }

    public void setRestEnabled(boolean restEnabled) {
        this.restEnabled = restEnabled;
    }
}
```

下面是fibonacci服务类，当前其计算代码总是返回-1，这旨在模拟一项未完成的功能：

```
@Service("fibonacci")
public class FibonacciService {

    public int getNthNumber(int n) {
        return -1;
    }
}
```

还需要一个用于存储计算结果的包装器：

```
public class FibonacciNumber {
    private final int number, value;

    public FibonacciNumber(int number, int value) {
        this.number = number;
        this.value = value;
    }

    public int getNumber() {
        return number;
    }

    public int getValue() {
        return value;
    }
}
```

下面是负责处理fibonacci服务查询的fibonacciRESTController类：

```
@RestController
public class FibonacciRestController {
    @Autowired
```

```
    FibonacciFeatureConfig fibonacciFeatureConfig;

    @Autowired
    @Qualifier("fibonacci")
    private FibonacciService fibonacciProvider;

    @RequestMapping(value = "/fibonacci", method = GET)
    public FibonacciNumber fibonacci(
            @RequestParam(
                    value = "number",
                    defaultValue = "0") int number) {
        if (fibonacciFeatureConfig.isRestEnabled()) {
            int fibonacciValue = fibonacciProvider
                    .getNthNumber(number);
            return new FibonacciNumber(number, fibonacciValue);
        } else throw new UnsupportedOperationException();
    }

    @ExceptionHandler(UnsupportedOperationException.class)
    public void unsupportedException(HttpServletResponse response)
            throws IOException {
        response.sendError(
                HttpStatus.SERVICE_UNAVAILABLE.value(),
                "This feature is currently unavailable"
        );
    }

    @ExceptionHandler(Exception.class)
    public void handleGenericException(
            HttpServletResponse response,
            Exception e) throws IOException {
        String msg = "There was an error processing " +
                "your request: " + e.getMessage();
        response.sendError(
                HttpStatus.BAD_REQUEST.value(),
                msg
        );
    }
}
```

注意，方法`fibonacci`负责核实应当启用还是禁用`fibonacci`服务。如果该禁用，就引发`UnsupportedOperationException`异常。还有两个错误处理函数:第一个处理异常`UnsupportedOperationException`，第二个处理通用异常。

所有组件都就绪后，只需执行Gradle命令`run`：

```
$> gradle run
```

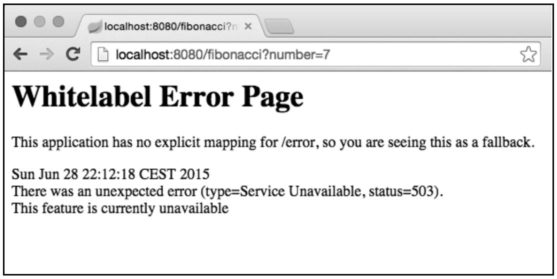
命令将启动在地址http://localhost:8080处搭建服务器的过程，如下面的控制台输出所示：

```
...
2015-06-19 03:44:54.157  INFO 3886 --- [           main] o.s.w.s.handler.
SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto
handler of type [class org.springframework.web.servlet.resource.
ResourceHttpRequestHandler]
```

```
2015-06-19 03:44:54.160  INFO 3886 --- [           main]
o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**]
onto handler of type [class org.springframework.web.servlet.resource.
ResourceHttpRequestHandler]
2015-06-19 03:44:54.319  INFO 3886 --- [           main] o.s.w.s.handler.
SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto
handler of type [class org.springframework.web.servlet.resource.
ResourceHttpRequestHandler]
2015-06-19 03:44:54.495  INFO 3886 --- [           main] o.s.j.e.a.Annota
tionMBeanExporter        : Registering beans for JMX exposure on startup
2015-06-19 03:44:54.649  INFO 3886 --- [           main] s.b.c.e.t.Tomcat
EmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2015-06-19 03:44:54.654  INFO 3886 --- [           main] c.p.tddjava.
ch09.Application             : Started Application in 6.916 seconds (JVM
running for 8.558)
> Building 75% > :run
```

启动应用程序后，使用浏览器执行查询。查询的URL为http://localhost:8080/fibonacci?number=7。

这个查询的输出如下所示。



从中可知，收到的错误对应于REST API在功能被禁用时发送的错误。如果功能被启用，返回的结果将为`-1`。

9.3.1 实现 `fibonacci` 服务

大多数读者可能都熟悉斐波那契数列，但有些读者可能不知道它为何物，下面进行简要介绍。

> 斐波那契数列是一个整数数列，通过反复使用公式$f(n)=f(n-1)+f(n-2)$计算得到。这个数列开头的两个数为$f(0)=0$和$f(1)=1$，而其他所有数字都是这样计算得到的：递归使用前面的公式，知道公式中的每项都为已知值0或1。
>
> 换言之，斐波那契数列为**0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…**
>
> 有关斐波那契数列的更详细信息，请参阅http://www.wolframalpha.com/input/?i=fibonacci+sequence。

我们还想限制计算斐波那契数所需的时间，为此对输入进行限制：fibonacci服务只计算斐波那契数列中第0~30个数字。

对于计算斐波那契数的类，一种可能的实现如下所示：

```
@Service("fibonacci")
public class FibonacciService {
    public static final int LIMIT = 30;

    public int getNthNumber(int n) {
        if (isOutOfLimits(n) {
        throw new IllegalArgumentException(
        "Requested number must be a positive " +
           number no bigger than " + LIMIT);
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        int first, second = 1, result = 1;
        do {
            first = second;
            second = result;
            result = first + second;
            --n;
        } while (n > 2);
        return result;
    }

    private boolean isOutOfLimits(int number) {
        return number > LIMIT || number < 0;
    }
}
```

为简洁起见，这里的演示未包含"红灯–绿灯–重构"这个TDD过程，但它贯穿整个开发过程。这里只列出最终的实现和测试：

```
public class FibonacciServiceTest {
    private FibonacciService tested;
    private final String expectedExceptionMessage =
        "Requested number " +
            "must be a positive number no bigger than " +
            FibonacciService.LIMIT;

    @Rule
    public ExpectedException exception = ExpectedException.none();

    @Before
    public void beforeTest() {
        tested = new FibonacciService();
    }

    @Test
    public void test0() {
        int actual = tested.getNthNumber(0);
```

```
        assertEquals(0, actual);
    }

    @Test
    public void test1() {
        int actual = tested.getNthNumber(1);
        assertEquals(1, actual);
    }

    @Test
    public void test7() {
        int actual = tested.getNthNumber(7);
        assertEquals(13, actual);
    }

    @Test
    public void testNegative() {
        exception.expect(IllegalArgumentException.class);
        exception.expectMessage(is(expectedExceptionMessage));
        tested.getNthNumber(-1);
    }

    @Test
    public void testOutOfBounce() {
        exception.expect(IllegalArgumentException.class);
        exception.expectMessage(is(expectedExceptionMessage));
        tested.getNthNumber(31);
    }
}
```

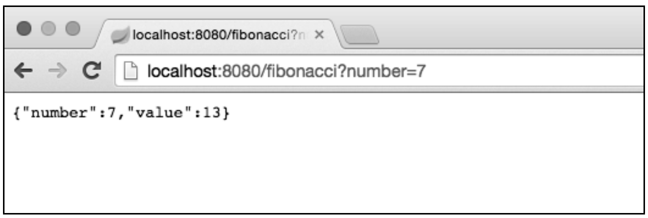
现在可以在文件`application.yml`中启用斐波那契数列计算功能，并使用浏览器执行一些查询以查看结果：

```
features:
    fibonacci:
        restEnabled: true
```

执行Gradle命令run：

```
$>gradle run
```

现在可以使用浏览器对REST API进行全面测试——要求计算第N个斐波那契数（$0 \leqslant N \leqslant 30$）。



N大于30时，返回的结果不再是数字，而是错误消息。



9.3.2　使用模版引擎

我们现在能够开关斐波那契数计算功能，但很多其他情形下，功能开关也很有用。比如对于链接到未完成功能的Web链接，可将其隐藏。这是一种很有趣的用法，因为我们可使用该链接的URL测试发布到生产环境的功能，同时对其他用户隐藏这个链接——想隐藏多久就隐藏多久。

为演示这一点，我们将使用前面说到的框架Thymeleaf创建一个简单的网页。

首先，添加一个新的控制标志：

```
features:
    fibonacci:
        restEnabled: true
        webEnabled: true
```

接下来，在一个配置类中映射这个新标志：

```
    private boolean webEnabled;
    public boolean isWebEnabled() {
        return webEnabled;
    }

    public void setWebEnabled(boolean webEnabled) {
        this.webEnabled = webEnabled;
    }
```

我们将创建两个模版。一个是主页，包含几个用于计算不同斐波那契数的链接。这些链接应仅在斐波那契数计算功能被启用时才可见，因此有一个模拟这种行为的可选块：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
    <title>HOME - Fibonacci</title>
</head>
<body>
<div th:if="${isWebEnabled}">
    <p>List of links:</p>
    <ul th:each="number : ${arrayOfInts}">
        <li><a
```

```
            th:href="@{/web/fibonacci(number=${number})}"
            th:text="'Compute ' + ${number} + 'th fibonacci'">
        </a></li>
    </ul>
</div>
</body>
</html>
```

第二个模版显示计算得到的斐波那契数，还包含一个返回到主页的链接：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
    <title>Fibonacci Example</title>
</head>
<body>
<p th:text="${number} + 'th number: ' + ${value}"></p>
<a th:href="@{/}">back</a>
</body>
</html>
```

这两个模版要发挥作用，必须位于特定位置，分别为`src/main/resources/templates/home.html`和`src/main/resources/templates/fibonacci.html`。

最后是核心部分——将前面所说的一切关联起来并使其正常工作的控制器：

```
@Controller
public class FibonacciWebController {
    @Autowired
    FibonacciFeatureConfig fibonacciFeatureConfig;

    @Autowired
    @Qualifier("fibonacci")
    private FibonacciService fibonacciProvider;

    @RequestMapping(value = "/", method = GET)
    public String home(Model model) {
        model.addAttribute(
            "isWebEnabled",
            fibonacciFeatureConfig.isWebEnabled()
        );
        if (fibonacciFeatureConfig.isWebEnabled()) {
            model.addAttribute(
                "arrayOfInts",
                Arrays.asList(5, 7, 8, 16)
            );
        }
        return "home";
    }

    @RequestMapping(value ="/web/fibonacci", method = GET)
```

9

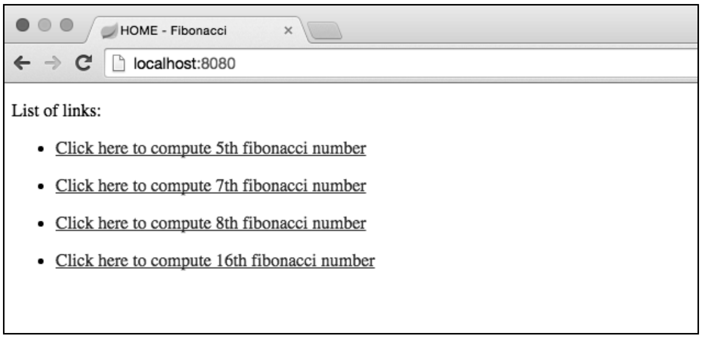
```
    public String fibonacci(
            @RequestParam(value = "number") Integer number,
            Model model) {
        if (number != null) {
            model.addAttribute("number", number);
            model.addAttribute(
                "value",
                fibonacciProvider.getNthNumber(number));
        }
        return "fibonacci";
    }
}
```

注意，这个控制器与前面的REST API示例中的控制器有点像，因为它们是使用同一个框架创建的，使用的资源也相同。然而，二者也存在一些细微差别，其中之一就是这里使用的注解为`@Controller`，而不是`@RestController`。因为Web控制器负责向模版页面提供自定义信息，而REST API负责生成用JSON对象表示的响应。

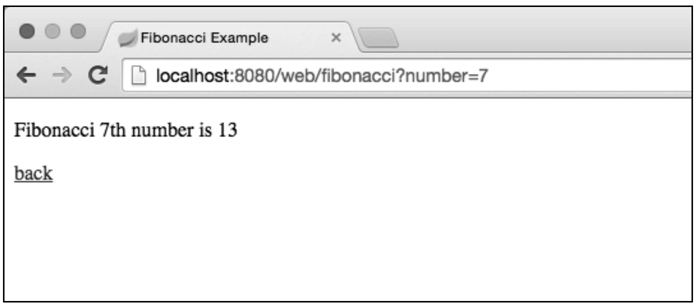
下面再次使用Gradle命令查看结果：

```
$> gradle clean run
```

这个命令生成主页。



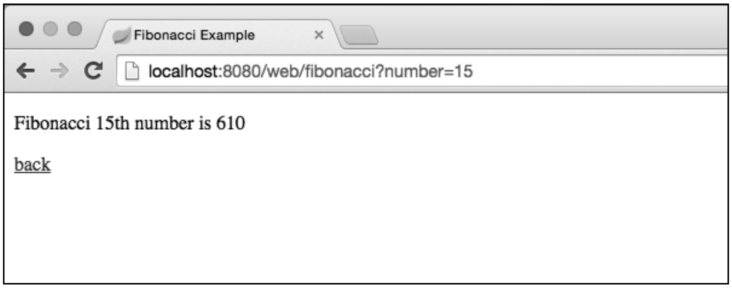
下面是单击第二个链接的结果。



我们使用如下代码关闭这项功能：

```
features:
    fibonacci:
        restEnabled: true
        webEnabled: false
```

再重新启用这个应用程序并浏览器主页，此时看不到链接，但依然能够计算斐波那契数，条件是知道相应的URL。如果手动输入URL http://localhost:8080/web/fibonacci?number=15，依然能够访问包含响应的相应页面。



这种做法很有用，但通常会无谓地增加代码复杂度。请别忘了重构代码，删除不再使用的旧开关，让代码更整洁、可读性更高。另外，一个不错的主意是，让修改无需重启应用程序就能生效；很多存储方式都不要求重启应用程序，其中最流行的是数据库。

9.4 小结

功能开关为在生产环境中隐藏和处理未完成的功能提供了不错的途径。根据需要将代码部署到生产环境时，使用功能开关好像有点怪；但需要持续集成、持续交付或持续部署时，经常使用功能开关。

我们简要介绍了功能开关，并讨论了其优缺点，还列举了功能开关很有用的一些典型情形。

很多库都可帮助我们实现功能开关，它们提供了大量功能，如使用Web界面处理功能、将首选项存储到数据库以及让你能够访问用户配置文件。

最后，我们实现了两种不同的方法：针对简单REST API的功能开关；在Web应用程序中使用功能开关。

第 10 章 综 述

10

“如果你总是重复过去做过的事情，结果也会一成不变。”

——阿尔伯特·爱因斯坦

我们介绍了大量理论，还进行了大量实践。整个旅程就像一辆高速列车，我们根本没有机会重温学到的知识，也找不到休息的时间。

好消息是现在终于找到了复习的空隙，我们将总结学到的所有知识，并介绍TDD最佳实践。其中有些最佳实践在前面提到过，但还有一些是新的。

10.1 TDD 概要

“红灯–绿灯–重构”是TDD的支柱，将TDD过程分解为可重复的短暂周期，其中每个阶段的持续时间通常以分钟乃至以秒计。我们编写一个测试，确定它不能通过后，编写刚好能让它通过的实现代码，然后运行所有测试，并进入绿灯阶段。编写代码后不断对其重构，直到其质量到达我们期望的程度。在这个阶段，测试应始终能够通过。在重构阶段，既不能引入新功能，也不能编写新测试。如此短的时间内完成所有这些任务好像不太可能，也可能让人感到担心。但愿通过前面的练习，你的技能、信心和速度都有所提高。

虽然TDD包含“测试”一词，但这并不是TDD带来的主要好处，也不是其目标所在。TDD首先是一种更佳的代码设计方式，而测试只是副产品，用来不断核实应用程序确实像期望的那样工作。

前面反复提到了速度的重要性。要确保速度，你需要更加熟悉TDD，同时别忘了使用测试替身（模拟对象、存根、间谍等）。使用测试替身可避免使用外部依赖，如数据库、文件系统、第三方服务等。

TDD还有哪些好处呢？文档是其中之一。鉴于只有代码能够准确而实时地呈现当前开发的应用程序，因此，你想更深入地了解某段代码的作用时，首先应求助于使用TDD编写的规范（也是代码）。

设计呢？使用TDD编写的代码设计得更好。使用TDD时，不用预先定义设计，相反，不断编写并实现规范的过程中，设计通常会变得清晰。与此同时，易于测试的代码都是设计良好的，因为测试要求我们必须应用一些最佳编码实践。

我们还了解到，TDD并非只适用于小型单元（方法），它也可用于更高层面。这些层面专注于功能或行为，可能横跨多个方法、类乃至应用程序和系统。在这些层面，使用的TDD是**行为驱动开发**（**BDD**）。不像TDD那样基于由开发人员为开发人员编写的单元测试，BDD可供组织的任何人使用。BDD涉及的是行为，而且是使用自然（普通）语言编写的。因此测试人员、业务代表等都能参与其创建，还可将其作为参考。

我们将遗留代码定义为不带测试的代码。我们遇到过遗留代码带来的一些挑战，并学习了一些让遗留代码可测试的技巧。

简要总结TDD后，下面归纳TDD最佳实践。

10.2　最佳实践

编码最佳实践是软件开发领域长期以来总结的一套非正式规则，可帮助改善软件质量。无论编写什么应用程序，都需要有一定创意（毕竟，我们希望打造新颖或更好的东西），而编码实践可帮助我们避免前人遇到过的一些问题。如果你刚接触TDD，最好遵循别人总结的部分乃至全部最佳实践。

我们将测试驱动开发最佳实践分为四类：

- 命名约定；
- 流程；
- 开发实践；
- 工具。

你将看到，这些最佳实践并非只适用于TDD。测试驱动开发中，很大一部分工作是编写测试，因此接下来将介绍的很多最佳实践也适用于测试，还有一些也适用于一般性编码。不管源自何处，你践行TDD时，这些最佳实践都很有帮助。

接受建议时请保留一点点怀疑态度。伟大的程序员不仅知道如何编码，还能够做出判断，确定哪些实践、框架和风格最适合其项目和团队。敏捷不是全盘接受他人制定的规则，而是知道审时度势，为团队和项目选择最合适的工具和实践。

10.2.1　命名约定

命名约定有助于更好地组织测试，从而让开发人员更容易测试。另一个好处是，很多工具都

要求遵循这些约定。大家使用的命名约定很多，这里介绍的只是沧海一粟。无论什么命名约定，都聊胜于无。最重要的是，团队的每个成员都知道要遵循哪些命名约定，并能熟练使用。选择流行的命名约定的优点在于，新加入团队的成员能快速掌握，因为既有的知识可提供帮助。

将实现代码和测试代码分开

好处：可避免不小心将测试和产品二进制文件一起打包；很多构建工具都要求测试位于特定源代码目录。

常见的做法是至少创建两个源代码目录，将实现代码放在目录`src/main/java`中，并将测试代码放在目录`src/test/java`中。在较大的项目，源代码目录可能更多，但也必须将实现代码和测试代码分开。

Gradle和Maven等构建工具不仅要求将测试代码和实现代码放在不同的源代码目录，还要遵循特定的命名约定。

你可能注意到，本书始终使用的文件`build.gradle`没有明确指定要测试什么，也没有指定要使用哪些类创建`.jar`文件。Gradle假定测试位于目录`src/test/java`，而要打包到jar文件的实现代码位于目录`src/main/java`。

将测试类和实现放在一个包中

好处：知道测试和代码位于同一个包中有助于更快找到代码。

正如前一个实践指出的，测试和代码虽然位于同一个包，但位于不同的源代码目录。

本书所有练习都遵循了这个约定。

以类似于受测类的方式给测试类命名

好处：知道测试和受测类的名称类似后，有助于快速找到受测类。

一种常见的做法是，也这样给测试命名，即为实现类加上后缀`Test`。例如，如果实现类为`TickTackToe`，就将测试类命名为`TickTackToeTest`。

然而，除贯穿重构练习都使用的测试类外，我们都使用后缀`Spec`。这有助于明确指出，创建测试方法的主要目的是指定要开发的内容。测试是规范的绝妙副产品。

给测试方法指定描述性名称

好处：有助于明白测试的目标。

使用对测试进行描述的方法名时，有助于掌握测试失败的原因，以及在什么情况下通过添加测试可提高代码覆盖率。必须明确指出测试前设置了哪些条件、测试将执行哪些操作以及期望的结果。

给测试方法命名的方式有很多，我们选择的方式是采用BDD场景中使用的`Given/When/Then`语法：`Given`部分描述前置条件，`When`部分描述操作，而`Then`部分描述期望的结果。如果测试没有前置条件（这通常是在用`@Before`和`@BeforeClass`注解的方法中设置的），可省略`Given`部分。

下面看一个为“井字游戏”创建的规范：

```
@Test
public void whenPlayAndWholeHorizontalLineThenWinner() {
    ticTacToe.play(1, 1); // X
    ticTacToe.play(1, 2); // O
    ticTacToe.play(2, 1); // X
    ticTacToe.play(2, 2); // O
    String actual = ticTacToe.play(3, 1); // X
    assertEquals("X is the winner", actual);
}
```

只要阅读这个方法的名称，就能知道它是做什么的：玩家落子后，如果其棋子填满了整条水平线，该玩家就赢了。

> 不要完全依赖注释以提供有关测试目标的信息。因为从IDE执行测试时，注释不会出现，它们也不会出现在CI或构建工具生成的报告中。

10.2.2 流程

TDD流程是一套最重要的实践。要成功实施TDD，有赖于本节介绍的实践。

> **先编写测试，再编写实现代码**
> 好处：这可确保编写的代码是可测试的，即每个代码都有为之编写的测试。

通过先编写或修改测试，开发人员将在着手编写实现代码前专注于需求，这是TDD与完成实现后再编写测试的主要差别所在。先编写测试的另一个好处时，可避免测试变成质量检查（而不是质量保证）的手段。我们要力图确保质量是内生的，而不是事后再去检查是否达到了质量标准。

> **仅在测试失败后才编写新代码**
> 好处：这确认了在没有实现的情况下，测试不管用。

如果测试不要求编写或修改实现就能通过，则说明要么它测试的功能已经实现，要么测试本身存在缺陷。如果测试定义的新功能没有实现而总是能够通过，就说明它毫无用处。测试必须因预期的原因而失败，此时，虽然不能保证它验证了正确的事情，但能够确定验证本身是正确的。

> **每次修改实现代码后，都再次运行所有测试**
> 好处：确保代码更改没有带来任何意料之外的副作用。

每次修改实现代码后，都应运行所有测试。理想情况下，测试的执行速度很快，且开发人员可在本地执行。将代码提交给版本控制系统后，应再次运行所有测试，以确保代码合并没有带来任何问题；这在多位开发人员协作开发代码时尤其重要。应使用诸如Jenkins（http://jenkins-ci.org/）、Hudson（http://hudson-ci.org/）、Travis（https:// travis-ci.org/）和Bamboo（https://www.atlassian.com/software/bamboo）等持续集成工具从仓库提取代码，对其进行编译并运行测试。

> **仅当所有测试都通过后才编写新测试**
> 好处：将始终专注于小型工作单元；实现代码几乎始终处于可运行的状态。

有时候开发人员很想编写多个测试再编写实现；还有些时候，开发人员对现有测试发现的问题置若罔闻，转而开发新功能。这些做法应尽可能避免。大多数情况下，违反前述规则都将增加技术债务，需要连本带息一起偿还。“实现代码几乎始终像期望的那样工作”是TDD的目标之一。有些项目由于交付日期迫在眉睫，或为避免超过预算而违反这条规则，将时间全部用于实现新功能，把与失败测试相关联的代码修复工作推后。这些项目通常不可避免地要推迟交付。

> **仅当测试都通过后才重构**
> 好处：这样的重构是安全的。

如果所有可能受重构影响的实现代码都有配套测试，且测试都通过，那么重构将是相对安全的。大多数情况下，重构期间都不需要编写新测试——对既有测试做细微修改即可。对重构来说，期望的结果是在修改代码之前和之后，所有测试都能通过。

10.2.3　开发实践

本节介绍的实践致力于以最佳方式编写测试。

> **编写让测试能够通过的最简单的代码**
> 好处：确保设计越来越清晰；避免实现不必要的功能。

这里的理念是，实现越简单，产品越好，且越容易维护。这个理念遵循了**KISS**原则。这个原则指出，对大多数系统而言，保持简单而不是复杂化的效果最好。因此设计的主要目标是简约，必须避免不必要的复杂性。

> **先编写断言，再编写操作**
> 好处：这将更早澄清测试目的。

编写断言后，测试目的就清晰了，这让开发人员可专注于满足该断言的代码，再专注于实际实现。

> **最大限度减少每个测试中的断言**
> 好处：避免不知道哪个断言导致测试失败；让更多断言得以执行。

如果一个测试方法中使用多个断言，可能难以判断哪个断言导致了测试失败。在持续集成过程中执行测试时，这种问题尤为常见。如果问题不能在开发人员的计算机中重现（例如，问题是环境因素导致的），可能难以修复，进而耗费大量时间。

断言失败时，其所属测试方法将停止执行。如果这个方法中还有其他断言，这些断言将不会执行，导致无法获得原本可用于调试的信息。

最后，包含多个断言会令人迷惑，不知道测试的目标到底是什么。

这种实践并不意味着每个测试方法都只能包含一个断言。如果有多个测试相同逻辑条件或功能单元的断言，可将它们都放在同一个测试方法中。

下面看几个示例：

```
@Test

public final void whenOneNumberIsUsedThenReturnValueIsThatSameNumber()
{
    Assert.assertEquals(3, StringCalculator.add("3"));
}

@Test
public final void whenTwoNumbersAreUsedThenReturnValueIsTheirSum() {
    Assert.assertEquals(3+6, StringCalculator.add("3,6"));
}
```

上述代码包含两个测试，它们明确指出了自己的目标。通过阅读方法名和查看测试，可以清楚地知道测试的是什么。请看下面示例：

```
@Test
public final void
```

```
whenNegativeNumbersAreUsedThenRuntimeExceptionIsThrown() {
    RuntimeException exception = null;
    try {
        StringCalculator.add("3,-6,15,-18,46,33");
    } catch (RuntimeException e) {
        exception = e;
    }
    Assert.assertNotNull("Exception was not thrown", exception);
    Assert.assertEquals("Negatives not allowed: [-6, -18]",
            exception.getMessage());
}
```

这个测试包含多个断言，但它们测试的是同一个逻辑功能单元。第一个断言确认存在异常，第二个断言确认异常消息是正确的。在同一个测试方法中使用多个断言时，这些断言都应包含对失败原因进行解释的消息，这样调试失败的断言将更容易。测试方法只有一个断言时，消息是受欢迎的，但并非必不可少，因为从方法名就能清楚知道测试目标。

```
@Test
public final void whenAddIsUsedThenItWorks() {
    Assert.assertEquals(0, StringCalculator.add(""));
    Assert.assertEquals(3, StringCalculator.add("3"));
    Assert.assertEquals(3+6, StringCalculator.add("3,6"));
    Assert.assertEquals(3+6+15+18+46+33,
           StringCalculator.add("3,6,15,18,46,33"));
    Assert.assertEquals(3+6+15, StringCalculator.add("3,6n15"));
    Assert.assertEquals(3+6+15,
           StringCalculator.add("//;n3;6;15"));
    Assert.assertEquals(3+1000+6,
           StringCalculator.add("3,1000,1001,6,1234"));
}
```

这个测试包含很多断言，其目标不明；如果其中一个断言失败，将无法知道其他断言是否管用。使用CI工具执行这个测试时，如果它失败，可能难以搞明白失败原因。

> **不要让测试依赖其他测试**
> 好处：测试能以任何顺序独立执行，不管运行全部还是部分测试，都将如此。

每个测试都应独立于其他测试。开发人员应该能够执行任一、部分或全部测试。鉴于测试运行器的设计，测试的执行顺序通常是不确定的。如果测试之间存在依赖关系，引入新测试时，这种依赖关系很容易遭到破坏。

> **测试的运行速度必须很快**
> 好处：这样就能经常运行测试。

如果测试的运行时间很长，开发人员将不再运行，或者只运行与所做修改相关的很少一部分测试。除促使开发人员使用它们外，测试运行速度快的另一个好处是可快速提供反馈。问题发现

得越早，修复就越容易，因为此时对导致问题的代码还记忆犹新。如果等待测试执行完毕期间，开发人员已开始着手处理下一个功能，他可能决定将修复问题的工作推迟到完成新功能后去做。另一方面，如果他放下手头的工作去修复问题，将因调整思路而浪费时间。

测试的运行速度必须很快，让开发人员每次修改后都能运行所有测试，而不会感到厌烦或气馁。

使用测试替身
好处：减少代码依赖并提高测试的执行速度。

要快速执行测试并专注于单个功能单元，必须使用模拟对象。通过模拟受测方法的外部依赖，开发人员能够专注于手头的任务，而无需花时间建立这些依赖。在小组较大或多个小组协同工作的情况下，这些依赖甚至可能还没有开发。另外，不使用模拟对象的情况下，测试的执行速度通常很慢。数据库、其他产品、服务等都非常适合使用模拟对象进行代替。

Use set-up and tear-down methods
好处：让我们能够在类或各个测试方法之前和之后执行设置和拆除代码。

我们通常需要在测试类或其每个方法之前执行一些代码，为此，JUnit提供了注解`@BeforeClass`和`@Before`。注解`@BeforeClass`在类被加载前（第一个测试方法运行前）执行与之关联的方法，而`@Before`在每个测试运行前执行与之关联的方法。这两个注解用于测试存在前置条件的情形，最常见的例子是在数据库（很可能位于内存）中设置测试数据。

与这两个注解对应的是`@After`和`@AfterClass`，它们的主要用途是销毁设置阶段或其他测试创建的数据或状态。前一个实践说过，每个测试都应独立于其他测试；另外，任何测试都不应受其他测试的影响。拆除阶段有助于确保系统就像之前未执行任何测试一样。

不要在测试中使用基类
好处：让测试更清晰。

开发人员常常像编写实现代码那样编写测试代码。一种常见的错误是创建基类，并让测试对其进行扩展。这种做法可避免代码重复，但代价是测试不清晰。应尽可能在测试中少用甚至不用基类。如果为理解测试背后的逻辑而必须从测试类导航到其父类、祖父类等，通常会引发无谓的迷惑。相比于避免代码重复，测试的清晰度更重要。

10.2.4　工具

TDD中，编码和测试都严重依赖其他工具和流程。这里列出一些最重要的，其中每个都是庞大的主题，无法在本书中深入探索，只做简要的描述。

代码覆盖率和持续集成（CI）工具
好处：确保测试覆盖每个角落。

为判断是否测试了所有代码、分支以及复杂度，代码覆盖率实践和工具很有帮助。这样的工具包括JaCoCo（http://www.eclemma.org/jacoco/）、Clover（https://www.atlassian.com/software/clover/overview和Cobertura（http://cobertura.github.io/cobertura/）。

除非项目非常简单，否则**持续集成**（**CI**）必不可少。最常用的持续集成工具包括Jenkins（http://jenkins-ci.org/）、Hudson（http://hudson-ci.org/）、Travis（https://travis-ci.org/）和Bamboo（https://www.atlassian.com/software/bamboo）。

结合使用TDD和BDD
好处：涵盖面向开发人员的单元测试和面向客户的功能测试。

虽然TDD单元测试很好，但很多情况下，这并不能提供项目需要的所有测试。TDD可提高开发速度，帮助完成设计，并通过快速反馈赋予开发人员以信心；而BDD更适合用于集成测试和功能测试，它提供了更佳的需求收集流程（通过叙述收集），还是一种更佳的沟通方式（通过场景与客户沟通）。你应同时使用它们，因为它们一起提供了完整的流程，让所有相关方和团队成员都参与其中。你应结合使用（基于单元测试的）TDD和BDD驱动开发过程。我们推荐使用TDD提高代码覆盖率以及提供快速反馈，使用BDD自动化验收测试。TDD主要致力于白盒测试，BDD通常致力于黑盒测试，但它们都专注于质量保证而不是质量检查。

10.3　这只是开始

你可能指望读完本书后就能知道测试驱动开发的方方面面，如果真是这样，很抱歉，让你失望了。要掌握任何手艺都需要很长的时间和大量的实践，TDD也不例外。请继续努力吧，将学到的知识用于项目，与同事分享这些知识，最重要的是实践、实践再实践。就像空手道一样，要想全面掌握测试驱动开发，只能通过持续练习，别无他法。我们使用TDD很长时间了，依然常常面临新的挑战，也常常学到改进手艺的新方式。

10.4　这并非终点

本书的编写是个漫长的旅程，期间充满了艰难险阻，但我们很享受这个过程，但愿你也觉得阅读本书是一种享受。

我们通过博客（http://technologyconversations.com）分享有关各种主题的经验。

- Java开发必读
- 从使用TDD开始，改善设计和代码的质量、简化重构工作、提高代码覆盖率

- 卓有成效地践行测试驱动开发所需的工具和框架
- 高效执行“红灯-绿灯-重构”过程
- 如何以独立于其他代码的方式进行有效的单元测试
- 使用各种技巧设计简单而易于维护的代码
- 使用模拟框架和技巧轻松编写测试并快速执行
- 结合行为驱动开发和单元测试进行TDD
- 重构遗留代码



[PACKT]
PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/程序设计/Java
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-46501-6
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：
ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks